System Level Modelling and Performance Estimation of Embedded Systems

Tranberg-Hansen, Anders Sejer

Publication date:
2011

Document Version
Publisher's PDF, also known as Version of record

System Level Modelling and Performance Estimation of Embedded Systems

Anders Sejer Tranberg-Hansen

Kongens Lyngby 2011
IMM-PHD-2011-248

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

This thesis was prepared at the Technical University of Denmark in partial fulfillment of the requirements for acquiring the Ph.D. degree. The project has been carried out at the Embedded Systems Engineering Section of the Department of Informatics and Mathematical Modelling, supervised by Professor Jan Madsen.

Espergærde, January 2011

Anders Sejer Tranberg-Hansen

Summary

The advances seen in the semiconductor industry within the last decade have brought the possibility of integrating evermore functionality onto a single chip forming functionally highly advanced embedded systems. These integration possibilities also imply that as the design complexity increases, so does the design time and effort. This challenge is widely recognized throughout academia and the industry and in order to address this, novel frameworks and methods, which will automate design steps as well as raise the level of abstraction used to design systems, are being called upon. To support an efficient system level design methodology, a modelling framework for performance estimation and design space exploration at the system level is required.

This thesis presents a novel component based modelling framework for system level modelling and performance estimation of embedded systems. The framework is simulation based and allows performance estimation to be carried out throughout all design phases ranging from early functional to cycle accurate and bit true descriptions of the system, modelling both hardware and software components in a unified way. Design space exploration and performance estimation is performed by having the framework produce detailed quantitative information about the system model under investigation.

The project is part of the national Danish research project, Danish Network of Embedded Systems (DaNES), which is funded by the Danish National Advanced Technology Foundation. The project is carried out in collaboration with the Danish company and DaNES partner, Bang & Olufsen ICEpower. Bang & Olufsen ICEpower provides industrial case studies which will allow the proposed modelling framework to be exercised and assessed in terms of ease of use, production speed, accuracy and efficiency.

The framework allows a given embedded system to be constructed and explored

before a physical realization is present and it can be used in the design of completely new systems or for modification of legacy systems. The primary benefits of the framework are the possibilities of exploring a large number of candidate systems within a short time frame leading to better designs, easier design verification through an iterative refinement of the executable system description, and finally the possibility of a reduction of the time-to-market of the design and implementation of the system under consideration.

In practice, however, additional time spent on software development in order to provide commercial quality tools supporting the method is required.

Resumé

Inden for de sidste årtier har halv-lederindustriens fremskridt medført, at det kan lade sig gøre at integrere mere og mere funktionalitet på en enkelt chip, hvilket har gjort det muligt at frembringe funktionelt stadigt mere avancerede indlejrede systemer. Disse integrationsmuligheder har dog samtidig haft den konsekvens, at når designkompleksiteten øges, så øges designtiden og indsatsen tilsvarende. Denne udfordring er bredt anerkendt i såvel den akademiske verden som i industrien, og for at imødegå denne kræves nye innovative værktøjer og metoder, som vil gøre det muligt at automatisere de enkelte designfaser, og som samtidig kan hæve det abstraktionsniveau, hvorved systemer designes. For at understøtte effektive systemniveaudesign-metodologier kræves et modelleringsframework, som muliggør performanceestimering og designrumsudforskning på systemniveau.

Denne afhandling præsenterer et nyt komponentbaseret modelleringsframework til brug i systemniveaumodellering og performanceestimering af indlejrede systemer. Frameworket er simuleringsbaseret og muliggør performanceestimering gennem alle designfaser, fra tidlige funktionelle beskrivelser til detaljerede beskrivelser med korrekt datamodellering og tidsmæssig opførsel af systemet, som modellerer både software og hardware i samme model. Designrumsudforskning og performanceestimering udføres ved at lade frameworket producere detaljerede kvantitative informationer om den systemmodel, som betragtes.

Projektet er en del af det nationale danske forskningsprojekt, Danish Network of Embedded Systems (DaNES), som støttes af Højteknologifonden. Projektet er udført i samarbejde med den danske virksomhed og DaNES partner, Bang & Olufsen ICEpower. Bang & Olufsen ICEpower har stillet industrielle casestudies til rådighed, som har gjort det muligt at afprøve det præsenterede modelleringsframework i praksis.

Frameworket gør det muligt at konstruere og udforske et givent indlejret system,

før det bliver fysisk realiseret, og det kan bruges til design af nye systemer såvel som til modifikation af eksisterende systemer. De primære fordele ved frameworket er muligheden for at udforske et stort antal systemkandidater inden for kort tid, hvilket gør det muligt at opnå bedre designs, nemmere verifikation igennem en iterativ raffinering af systembeskrivelsen og, sidst men ikke mindst, muligheden for at opnå en reduktion i den samlede tid, der bruges på design og implementering af et givent system.

Yderligere tid til udvikling af software, således at der kan opnås værktøjer af kommerciel kvalitet til understøttelse af frameworket i praksis er dog påkrævet.

Preface

This thesis presents the work that has been done in the course of the Ph.D. project with the title System Level Modelling and Performance Estimation of Embedded Systems. The project has been carried out in the period from December 1st 2007 to November 30th 2010 and it is part of the national Danish research project Danish Network of Embedded Systems (DaNES) which is funded by the Danish National Advanced Technology Foundation. The project was carried out in collaboration with the Danish company and DaNES partner, Bang & Olufsen ICEpower. Bang & Olufsen ICEpower has provided industrial case studies allowing the work of the project to be exercised and assessed.

I would like to express my gratitude to my supervisor, Professor Jan Madsen for letting me have this opportunity and for thought inspiring discussions and highly useful inputs throughout the project.

Also, I would like to thank Professor Alberto L. Sangiovanni-Vincentelli for hosting me during my stay at the University of California, Berkeley, California, U.S.A. as a visiting scholar in the fall of 2009.

Furthermore, as mentioned above, the project has been carried out in collaboration with Bang & Olufsen ICEpower to whom I am deeply indebted for letting me have the opportunity to combine elements from the academic and industrial worlds and, so, relate the theoretical work of the thesis to real world problems.

Finally I would like to thank my family for their endless love and support. In particular my beloved Sara for being extremely understanding and supportive during the many days and nights when I was working on this thesis and our twin sons August and William.

Scientific contributions

[1] Tranberg-Hansen, A. S., Madsen, J., and Jensen, B. S. 2008. A Service Based Estimation Method for MPSoC Performance Modelling. *International Symposium on Industrial Embedded Systems*, 43–50.

[2] Tranberg-Hansen, A. S. and Madsen, J. 2008. A Service Based Component Model for Composing and Exploring MPSoC Platforms. *International Symposium on Applied Sciences in Bio-Medical and Communication Technologies*, 1–5.

[3] Tranberg-Hansen, A. S. and Madsen, J. 2009. Exploration of a Digital Audio Processing Platform Using a Compositional System Level Performance Estimation Framework. *International Symposium on Industrial Embedded Systems*, 43–50.

[4] Tranberg-Hansen, A. S. and Madsen, J. 2009. A Compositional Modelling Framework for Exploring MPSoC Systems. In *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis.* New York, NY, USA: ACM, 2009, 1–10.

[5] Tranberg-Hansen, A. S. and Madsen, J. 2011. *Real-time Simulation Technologies: Principles, Methodologies, and Applications*, Chapter: A Service Based Simulation Framework for Performance Estimation of Embedded Systems. CRC Press.

Contents

Chapter 1

Introduction

Embedded systems are found everywhere in our daily lives from mobile phones and cars to home appliances. An embedded system is a dedicated computer system, most often composed of a mixture of both hardware and software elements, which are fully integrated in a given product, possibly hiding the actual computer system but enabling advanced features to e.g. the user of the product. The embedded system interacts with the environment and most often constraints on the interaction are imposed, implying that the system must behave and respond in a specific manner under one or more limiting factors such as the available power budget, requirements on the response time, the total cost of the system, etc.

The project presented in this thesis is part of the national Danish research project Danish Network of Embedded Systems (DaNES) which is funded by the Danish National Advanced Technology Foundation. DaNES is focusing on compositional model driven design of embedded systems with a strong industry oriented focus. DaNES is a joint research project between academia and industry in which academia is represented by Aalborg University, the Technical University of Denmark and the University of Southern Denmark. Industry is represented by a number of Danish companies working within areas as different as health care, automation and audio power conversion.

The current project presented in this thesis is carried out in collaboration with the Danish company and DaNES partner, Bang & Olufsen ICEpower. Bang & Olufsen ICEpower is a relatively young and innovative company, founded in 1999, focusing on audio power conversion within a number of markets ranging from mobile over consumer to professional and automotive.

1.1 Motivation

Since the 1960ies, the number of transistors in integrated circuits has grown exponentially, as predicted by Gordon Moore in 1965 [75], and transistor counts of an astonishing 3 billion in a single chip are now seen [90]. The advances of the semiconductor industry seen within the last decades have had a tremendous influence on embedded systems. A range of new opportunities have evolved such as the possibility of having entire systems consisting of multiple (or many) processors, memories and interfaces integrated onto a single chip. This has brought the possibility of integrating evermore functionality in embedded systems under constantly shrinking area and power budgets. The complexity of such systems is inherently high and thus the design of these is far from trivial taking time-to-market and development costs into consideration. As a consequence, the advances of the semiconductor industry and the associated integration possibilities have also had a strong impact on the design of embedded systems in the sense that as the design complexity increases, so, does the design time and effort.

Seen from the perspective of Bang & Olufsen ICEpower, the audio industry in general is experiencing a shift in the functionality provided which means that high-end functionality previously reserved for the professional market is now being introduced in consumer products due to the new possibilities that have evolved with the advances in the semiconductor industry. Thus mobile phones can be used as music players and are equipped with equalizers, smart bass, advanced dynamic protection systems and other traditionally advanced functionalities. Therefore, Bang & Olufsen ICEpower is experiencing an increasing demand for integrating digital audio processing capabilities in switching audio power conversion systems. This allows the implementation of algorithms used to e.g. improve the channel sound quality, protection of components or adding different feature level algorithms delivering e.g. sound field control. The requirements which the implementations must meet depend on the field of application, e.g. consumer, mobile, automotive etc., implying that a highly flexible design methodology for designing audio digital signal processing systems is needed offering an efficient implementation and a low design time due to tight time-to-market constraints.

The mixture of tight time-to-market constraints and an increasing design complexity associated with the design of embedded systems of today, not only at Bang & Olufsen ICEpower but throughout the industry, has meant that the majority of design methodologies have focused on reusability as a common denominator, whether this is achieved through the reuse of efficient hardwired function blocks with limited flexibility, or in the use of various flexible general purpose or domain specific processors (e.g. digital signal processors) with associated software tools which, in exchange for flexibility, sacrifice efficiency.

Flexibility and reusability of the individual components are central issues because the growing complexity of the individual design blocks implies that more time is needed to design each. To compensate for this, the designers strive to amortize design time on the largest possible volume, i.e. use the same block in as many products as possible.

Still, the tight time-to-market constraints coupled with the increasing design complexity makes the risk of sub-optimal implementations inherently evident as discussed for multi-processor system-on-chip systems in [70]. A major reason for this is the difficulty of getting feedback on the consequences of a design choice before the system has been realized physically or at least described at a very low level of abstraction. This makes the experience of the designers of the system a key element in the early design-phases and at the same time severely limits the possibilities of exploring the design space. Hence there is a great need for getting trustworthy feedback to the consequences of a design choice throughout the development phases no matter what design methodology is used.

These challenges are widely acknowledged throughout academia and the industry and, already a decade ago, were identified as the *design productivity gap* in [9]. In order to address this, novel frameworks and methods which will both automate design steps and raise the level of abstraction used to design systems are being called upon. Especially system level design has gained at lot of attention and is believed to be one way to address the increasing design-productivity gap. System level design is the design of complete systems based on a specification of the functional requirements to the system in combination with a number of constraints which the system has to fulfil, basically limiting the degrees of freedom from which the designer can choose. Most often system level design methods start out at a high level of abstraction, gradually refining the individual parts until a level suitable for implementation is reached.

In order to support an efficient system level design methodology, a modelling framework for performance estimation and design space exploration at the system level is required. As a consequence, in recent years much work, both in the industry and in academia, has focused on the development of frameworks and methods for estimating the performance at the system level [11, 13, 63, 86]. Performance estimation at system level, however, is not a trivial task and the level of accuracy obtained and the time used to produce the performance estimates are highly dependent on the level of abstraction which is used to describe the system. Intuitively, there is a trade-off between the speed at which performance estimates can be produced and the level of accuracy which is obtained. In general, the problem is tackled by many frameworks by allowing a gradual refinement of models, using a high level of abstraction for the early estimates, then lowering the level of abstraction, moving through the design space towards more detailed estimates and in this way performing the trade-off.

The work presented in this thesis takes it point of departure in the challenges

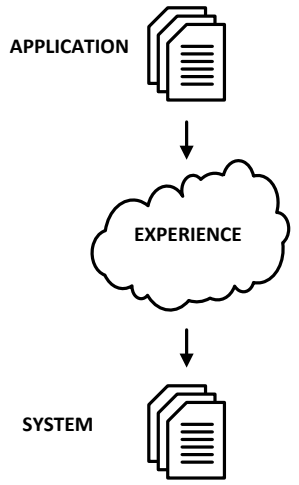


Figure 1.1: Abstraction of typical design flow.

faced by Bang & Olufsen ICEpower and there has been a strong focus on producing a modelling framework which will support performance estimation throughout the design phases. However, the generality of the work presented implies that it is by no means bound to the design methodology used at Bang & Olufsen ICEpower and that general applicability is preserved.

As is the case in the general industry, Bang & Olufsen ICEpower shares a wish of accomplishing lower design time for increasingly complex systems maintaining the high performance of the systems developed with the existing design flow. These goals must be met under a constantly shrinking area and power budget - especially if mobile platforms are targeted. To make this discussion at bit more concrete, a very condensed and overly simplified abstraction of the current design flow used at Bang & Olufsen ICEpower for designing audio processing applications is given in figure 1.1.

In general the functional requirements are translated into high level functional executable specification in the form of implementations of algorithms in e.g. Matlab or C/C++. These tools are excellent for capturing and exploring the functional aspects of the application. However, already at this stage, it is customary to have specific target architectures in mind, and thus critical design choices are already being made which limits the possible candidate architectures, in this way narrowing the system realization options significantly. Currently, reliable estimates of the performance of the system cannot be generated before the system is realized at a very low level of abstraction such as assembly level or C/C++ implementations of software parts executing on instruction set simulators, or for the dedicated hardware parts, as register transfer level de-

scriptions either through simulation or actual prototype implementations. This implies that radical design changes such as a different partitioning of the application is very hard to accomedate within the development cost budget and time-to-market constraints imposed on the system at this late stage in the design process.

For these reasons, the current design flow at Bang & Olufsen ICEpower has been considered tedious and extremely time consuming due to a number of limitations and barriers between the different layers of abstraction used to specify, design and implement a given application. However, most troublesome, as illustrated in figure 1.1, is the partitioning of the system into hardware and software parts, which is decided mainly based on experience and back-of-the-envelope calculations because no means currently exists which can help make the decision on a sound and well founded basis in the early stages of the design flow. The design flow used by Bang & Olufsen ICEpower and the problems faced is by no means unique and can basically be categorized as an example of a classical co-design problem found in a number of other companies as well. Today, the scope of the classical co-design problem of deciding which elements are to be implemented in hardware and which are to be implemented in software has been heavily extended due to the very heterogeneous nature of embedded systems of today.

1.2 Contributions

In order to address the lack of feedback to the consequence of a design choice and to provide a framework which can aid designers of a system throughout the development phases, this thesis presents, as one of the main contributions, a novel compositional framework for system level modelling and performance estimation of heterogeneous embedded systems. The framework is simulation based and allows performance estimation to be carried out throughout all design phases ranging from early functional to cycle accurate and bit true descriptions of the system. The simulation of a system model makes it possible to produce detailed information regarding the runtime properties of the specified system and, so, it can direct the designer to the elements which ought to receive special attention.

The framework presented, illustrated in figure 1.2, is related to what is known as the Y-chart approach [12, 50] which dictates a separate specification of the functional and implementation specific aspects, in our case represented by an *application model* and a *platform model* respectively, and relating the two through an explicit mapping step in order to allow these to be explored (partly) independently. However, in our case the application model is refined in its own iteration branch as step one, verifying the functionality of the application model only.

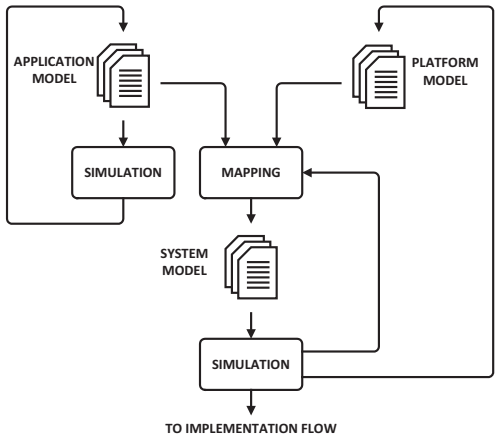


Figure 1.2: Overview of the framework.

When this step has been performed, the application model is left unchanged, and only the mapping and platform model is being refined in step two. If the application model needs to be changed, it implies that the functionality of the application has changed. Hence, a new iteration of step one is required in order to verify the new functionality before repeating step two.

The key strengths of the framework are the flexibility and refinement possibilities as well as the possibility of having components described at different levels of abstraction to co-exist and communicate within the same model instance. This is achieved by separating the specification of functionality, communication, cost and implementation (as advocated in [48]) and by using an interface based approach combined with the use of abstract communication channels. The interface based approach implies that component models can be seamlessly interchanged in order to investigate different implementations, possibly described at different levels of abstraction, constrained only by the requirement that the same interface must be implemented. Additionally, the use of component models allows the construction of component libraries with a high degree of reusability as a result.

A key concept in the framework is the notion of services which plays an important role in order to achieve a decoupling of the specification of functionality, communication, cost and implementation. A service is defined as a logical abstraction which represents a specific functionality or a set of functionalities offered by a component. In this way, services are used to abstract away the implementation details of the functionality which is offered by the component. Thus, the service abstraction allows two different models to offer the same services, having the same functional behaviour but with a different implementation, cost

and/or latency associated and, so, allow different implementations of a model to be investigated easily.

The notion of services allows a unified modelling approach for capturing system components using *service models*. The service model is another main contribution of the work carried out in this project and is the fundamental modelling component of the framework.

The functionality of the target application is captured by an application model. Application models are composed of a number of tasks each represented by a service model. The tasks of an application model serves as a functional specification of the application only specifying a partial order of service requests needed in order to capture the functionality of the application. No assumptions on who will provide the services required is made and, thus, the specification of functionality and implementation is separated.

The target architecture is modelled by a platform model and is composed of one or more service models. The service models of a platform model can be described at arbitrary levels of abstraction implying that in one extreme they only associate a cost with the execution of a service request or, in the other, the service request is modelled in the platform model both cycle accurate and bit true. Costs can be associated with service requests, either computed dynamically or pre-computed. It is the cost of the execution of a service which differentiates different implementations of the particular service.

Quantitative performance estimation is performed at system level through the simulation of a system model. A system model is constructed through an explicit mapping of the service models of an application model to the service models of a platform model. The service models of an application model, when executed, request the services offered by the service models of the platform model onto which they are mapped, modelling the execution of the requested functionality, taking the implementation specific details and required resources into consideration and associate a cost with each service requested. In this way, it becomes possible to associate a quantitative measure with a given system model and, hence, it becomes possible to compare systems and select the best suited one from a well-defined criteria.

The remaining part of this section strives to identify the main contributions of the work carried out during the course of this Ph.D. project and which will be presented in this thesis. The main contributions of this thesis are the following:

- A compositional service based framework for system level performance estimation. This includes the general concepts and a proof-of-concept implementation in Java [73], allowing the framework to be used for performance estimation in practice.

- The service model - a meta-model for unified hardware and software modelling supporting components described at multiple levels of abstraction, separation of behaviour and implementation, cost and communication as well as support for multiple models-of-computation to co-exist.
- A service based model-of-computation for modelling synchronous hardware components as an example of one model-of-computation which can be used in the presented system level modelling and performance estimation framework in order to capture synchronous hardware components.
- An industrial case study carried out at the Danish company Bang & Olufsen ICEpower. Throughout the thesis, examples have been taken from Bang & Olufsen ICEpower. Quite some time was spent on an industrial case study in order to asses the quality of the presented framework. In the case study, an audio processing platform for mobile devices was in focus and the goal was to explore different implementation options for a particular application. The case study provided valuable feedback and gave rise to a number of ideas for improvements.
- Investigations on the foundation of an architecture description language for describing synchronous hardware components and automatically generating fast simulation models based on the proposed service based model-of-computation for modelling synchronous hardware.
- Initial work on a framework for automated, multi-objective, design space exploration using evolutionary multi-objective algorithms. Fuelled by the experiences of the manual design space exploration carried out in the industrial case study, initial work on a framework for automated design space exploration based on multi-objective meta-heuristics, combined with the use of the system level modelling and performance estimation framework developed, shows very interesting perspectives.

1.3 Reading guideline

The body of information presented in this thesis is rather extensive and, so, this section presents the structure of the thesis and provides reading guidelines. The thesis is written in such a way that a linear path through the chapters should be possible. In order to group related chapters, the thesis has been divided into a number of parts.

Part I presents the system level modelling and performance estimation framework and consists of the chapters 2 to 4.

Chapter 2 introduces the field of system level modelling and performance estimation with a focus on related work relevant for the framework presented in

this thesis.

Before the actual framework is presented, the unified service based component model, the service model, is presented in chapter 3. The service model is the fundamental modelling component for capturing the elements of which a system is composed.

Chapter 4 presents the compositional framework for system level modelling and performance estimation of heterogeneous embedded systems which is among the main contributions of the work carried out during the course of this project. Through simulations of a system model the framework makes it possible to produce detailed performance estimates of the specified system. Performance estimation can be carried out throughout all design phases ranging from early functional to cycle accurate and bit true descriptions of the system. The key strengths of the framework are the flexibility and refinement possibilities as well as the possibility of having components described at different levels of abstraction to co-exist and communicate within the same model instance.

Part II focuses on the realization and usage of the system level modelling and performance estimation framework presented in part I. Part II is composed of chapters 5 to 7.

Chapter 5 presents the current realization of the system level modelling and performance estimation framework presented in chapter 4 and gives details on the underlying simulation engine which has been implemented.

Chapter 6 introduces a model-of-computation for modelling synchronous hardware components. The model-of-computation is based on Hierarchical colured petri nets (HCPN) [44] and uses the notion of service to represent the functionality offered by a component, fitting into the service model concept described in chapter 3. The chapter will introduce the basic concepts and describe how a number of optimizations can be performed to obtain fast simulation of models.

Chapter 7 describes an industrial case-study used to illustrate the concepts and the usage of the system level performance estimation framework presented in detail.

Part III discusses extensions to the system level modelling and performance estimation framework and consists of the chapters 8 and 9 only.

Chapter 8 introduces initial work on a architecture description language used to generate fast simulations models using model-of-computation introduced in chapter 6 for modelling synchronous hardware components.

Chapter 9 describes the initial investigations performed in order to extend the system level performance estimation framework with an automated design space exploration module. Through the use of a multi-objective optimization algo-

rithm based on a well-known meta-heuristic, a framework is presented in which design space exploration can be performed automatically.

Finally, in part IV, chapter 10 concludes the thesis and discusses the limitations of the framework as well as giving pointers to future work.

Part I

System Level Modelling and Performance Estimation

Chapter 2

Introduction to System Level Modelling and Performance Estimation

In this chapter an introduction to the field of system level modelling and performance estimation of embedded systems which includes references to related work will be given.

The diversity of elements of which an embedded system is most often composed implies that multi-disciplinary knowledge is required in order to design systems. In order to capture the behaviour of the complete system, including both hardware and software, different models-of-computation are often better suited for modelling the different parts of the system, possibly including the environment with which the system interacts. Methods and frameworks, which have support for multiple models-of-computation, are thus very beneficial, and might even be a requirement, in order for a system level modelling framework to become successful. However, system level modelling frameworks often target specific application domains in order to simplify the challenge of allowing multiple models-of-computation to co-exist within the same model instance. Quite a number of frameworks, e.g. target stream based applications only as seen in [46, 47], for which very well suited models-of-computation exist, e.g. Kahn process networks [45], Synchronous Dataflow [60], etc., and which cover the relatively large class of multi-media applications. Ptolemy II [28], on the other hand, is a framework developed specifically for supporting the co-existence of

multiple models-of-computation and a hierarchical component based approach is employed for modelling heterogeneous systems. Components are composed of a director and a number of actors. Actors communicate using tokens and fire regulated by a director. This approach allows different models-of-computation to be implemented. However, Ptolemy II is not developed for performance estimation of embedded systems specifically.

System level design typically starts with a decomposition of the application into concurrent elements which allows a mapping onto possibly multiple processing elements. Subsequently, a partitioning into hardware and software parts is required in order to achieve the targeted performance requirements. Having performed these challenging tasks, performance estimates of the resulting system model can now be produced, either through simulation or analytically, and the system can then be evaluated.

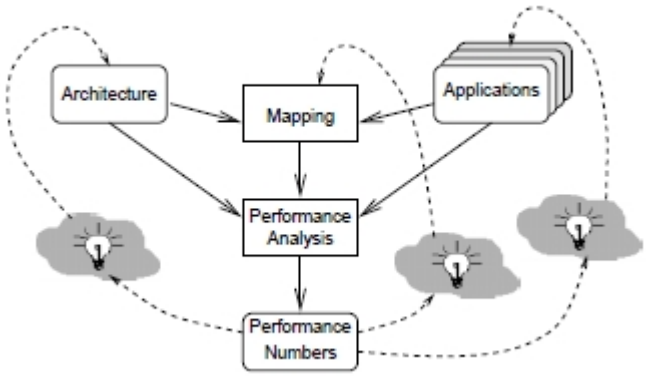


Figure 2.1: Illustration of the Y-chart [12, 50] appraoch, The figure is from [88], modified from the original in [50].

In the Y-chart approach [12, 50], which is a general approach used in the design and exploration of embedded systems, the application and architecture are specified individually, allowing both to be independent of each other. The application is then mapped to the architecture through an explicit mapping step relating the two descriptions. The result of the mapping step is a full system description which can then be evaluated. The separate specification of the application and candidate architecture allows the designer to experiment with different mappings, restructure the application or add or remove elements from the target architecture without imposing any impact on the remaining elements. The clear separation of the specification, into application and architecture, is a clear advantage and thus many frameworks and methods are based on varia-

tions of this approach e.g. [12, 15, 63, 72, 101]. The MESCAL methodology [72] e.g. presents a structured way of generating performance estimates of processor-centric systems, using a framework called Teepee which is based on the Y-chart methodology and uses Ptolemy II as the underlying simulation and analysis framework.

The separate specification of functionality and implementation, as used in the Y-chart methodology, is advocated even harder in [48], known as the concept of *separation of concerns*, which dictates separate specification of functional behaviour, communication, cost and implementation and is the foundation of the platform-based design paradigm [48, 92]. The concept of separation of concerns provides maximum freedom in the exploration of the design space by allowing applications to be mapped to arbitrary target architectures with the only constraint being that they must implement the required functionality but without restrictions on how this is implemented. In this way multiple architectures can be evaluated simply by changing the architecture model, letting the cost associated with the execution of the application on the particular platform be the differencing factor and, in this way, allowing the best suited platform to be selected.

The Metropolis [13] framework has been developed to support the platform based design principle [48, 92]. The framework provides an infrastructure for system level modelling by defining abstract semantics through the use of a meta-model based on the tagged signal model [59]. The meta-model allows the integration of heterogeneous components at different levels of abstraction. Fundamental to the framework is the use of the concept of orthogonalization/separation of concerns. Through an explicit mapping of the components of the functional model to the components of an architectural model, the framework supports iterative refinements from an initial specification to implementation. The Metropolis framework originates from the Polis [12] framework which was one of the first frameworks to use a separation of the specification of functionality and implementation.

The Polis and Metropolis frameworks are now being used as the basis of a new framework named Metro II [27]. The Metro II framework is similar to Metropolis in the sense that it focuses on the concept of platform based design; however, it is based on SystemC, and focuses especially on integrating third party IP blocks as opposed to the Metropolis framework which uses a proprietary language for specification of models.

The research within the field of system level design of embedded systems is tightly coupled with the research carried out within the hardware/software co-design domain, e.g. as described in [34, 71]. However, whereas early hardware/software co-design research focused on splitting the implementation into software running on a single processor and dedicated hardware functional units used as accelerators, the scope of the problem today has been extended to cov-

ering complex heterogeneous multi-processor platforms due to the constantly growing transistor counts of embedded systems of today. The arrival of multi-core systems years ago have already proved the big difficulty of utilizing such systems efficiently and, now, already at the brink of the many-core era, the problem seems ***not*** to be going away. Still the fundamental goal of the hardware/software co-design research is valid, and seems to be even harder to reach as the complexity of systems increase.

In general, the frameworks can be categorized as being based on simulation, analytical evaluation or a combination of the two. Several frameworks use simulations to obtain performance estimates and then use formal analysis for verifying different properties such as the validity of a given mapping, e.g. preventing deadlocks to occur.

Frameworks based on analytical methods of analysis are often very well suited for pruning the design space in the early design phases. However, due to restrictions on the level of abstraction used to model the components of a system, it often requires the use of simulation based methods in the latter design stages.

Simulation-based methods, on the other hand, are capable of spanning all abstraction levels from high level functional to cycle accurate and bit true simulations. This is the great strength of the simulation based methods; the Achilles heel of these frameworks, however, is that no guarantees can be given with respect to the performance estimates produced, i.e. they need not capture worst case conditions.

Hybrid approaches, in which the better of two worlds can be combined, seem to be the most interesting. However, in the future, as formal analysis methods mature, they might reduce the need for simulations even further.

In the following, an overview of the elements of system level performance estimation will be given, illustrated by references to existing research and practices. First, abstraction levels will be discussed followed by an overview of different approaches to capturing applications and architectures, how these are related to each other through a mapping step and finally how performance estimates can be generated.

2.1 Abstraction levels

As already stated, there is a close relationship between the level of abstraction used to describe a system and the quality of the performance estimates which can be produced. Also, the time and effort required to construct a system model and the time required to obtain the performance estimates are directly related to the level of abstraction used. This implies that choosing the right

level of abstraction for capturing a system is vital, and highly dependent on the goal of the modelling. When the final system is unknown, and designers are interested in finding the best suited architecture from a given set of constraints and requirements, it is often advantageous to have system descriptions at a higher level of abstraction in order to be able to search a larger part of the design space for the better solution. However, the main problem is to ensure the correct level of accuracy of the performance estimates produced so it is actually possible to base decisions on the generated performance estimates.

Generally, abstraction levels can be arbitrary and a mixing of abstraction levels within different domains, e.g. computation, communication and data, is often seen. In this discussion, the focus will be on timing accuracy of computation and communication and on data accuracy because these quantities are used in most design parameters and other quantities can often be derived in relation to these, e.g. power models, resource utilization, etc.

The highest level of abstraction is the functional level in which the purpose is to capture all functional requirements of the specification and allow these to be verified. Depending on the specification, this can be achieved using various models-of-computation suited for the specific domain or simply as a Matlab or C/C++ specification. In the general case, descriptions at this level of abstraction have no timing information incorporated and data types are most often standard generic data types offered by the host machine if defined at all.

Following the functional level of abstraction, a separation of the functional model into parallelly executable components is often seen. In this way communication requirements between components are becoming explicit. The partitioning of the application into smaller blocks allows designers to start exploring different implementation types for the individual components, performing initial hardware/software co-design investigations with the target architecture in mind. Quite often, this level of abstraction is based on abstract high level modelling of communication between components whereas the computing components are modelled purely functionally.

Gradually, refinements to various degrees of approximately timed models will lead to cycle accurate models arriving at the traditional design flow entry point of register transfer level models for the hardware parts.

In [82] an approach for system level design and performance estimation is presented targeted at processor-centric systems. The approach relies on an analytical evaluation of different configurations using neural networks in the early design phase. It is assumed that trained neural networks are available for all available processors that might be selected for the target architecture. As the design is refined, the best candidates from the high level analysis are selected and bus functional models are created and used for more detailed performance analysis based on register transfer level simulations of the IP components used.

These are assumed readily available in an IP library.

The concept of transaction level modelling (TLM) [16], have received severe interest throughout the last decade. The details of communication between computing components, is modelled not at signal level as done in register transfer level models but instead as communication transactions only. In this way e.g. a bus transfer can be modelled as a single transaction which can abstract away hand shaking, access control scheduling, etc. Also, an explicit separation of communication and computation is obtained. Channel models are used to connect computing components and allow these to communicate. Transactions are then implemented by requests to interface functions of the individual channel models. This abstraction, obviously, allow faster simulations to be obtained compared to register transfer level simulations. In practice, both SystemC [81] and SpecC [33] are languages which support the concept of transaction level modelling and the concept of channels.

SystemC [81] has been widely adopted for system level modelling and performance estimation throughout academia and industry. SystemC, however is not a framework for performance estimation in itself but provides the common syntax and semantics for describing systems consisting of both hardware and software at different levels of abstraction. Several groups are providing performance estimation frameworks which are based on SystemC - and with great success [27, 53, 68]. Also, with the introduction of the TLM 2.0 standard [80], the Open SystemC Initiative (OSCI) community seems to have gained a lot of support throughout industry and academia. TLM 2.0 defines common grounds for model interoperability between different developers by defining a set of abstraction levels and semantics for communication between models to which TLM 2.0 models must adhere and, combined with a definition of temporal decoupling of models, allows potential performance boosts in simulation speed.

In [52] a SystemC based design methodology is presented referred to as virtual architecture mapping in which a top down refinement process is applied. First, a functional model of the application is created and the functional properties verified. Then, the functional model is mapped onto a virtual architecture model in order to annotate timing characteristics. The high level virtual architecture model separates behaviour and timing by associating the behaviour of the system with a number of functional modules, and timing information with communication channels between modules. As a final step, co-verification is supported, using adapters for translating communication between modules described at different levels of abstractions.

In practice, many approaches rely on letting parts of the system be described at lower levels of abstractions in order to ensure better accuracy of the results of the vital components. In general, abstraction level refinement is an important element to support, in order to ease the usage of a system level performance estimation framework. Support for multiple levels of abstraction to co-exist

within the same model instance and a gradual refinement is seen in [11, 13, 84]. One of the main challenges when mixing abstraction levels is how to ensure correct communication between components described at different abstraction levels with a minimal effort from the designer and without having to change the component descriptions. One approach for automatic model refinement is presented in [84]. Another, more general approach, is to use adaptors for cross-abstraction level communication as suggested in [16].

In MESH [18], a framework is proposed focusing on modelling the execution of software on processors at a high level of abstraction. The goal is to bridge the gap between purely functional models and models based on the use of detailed instruction set simulators. MESH uses a thread based model, similar to the tagged signal model in [59] in which events are associated with a tag representing the time of the event and a value. Threads are then defined as a sequence of events. MESH distinguishes between physical and logical events. Logical events are events which cannot be physically related to a specific point of time. During runtime, logical events are resolved and associated with a physical point in time for execution which is implementation specific.

2.2 Application Modelling

The objective of application modelling is to capture the functionality of the application under investigation and obviously this can be done at many different levels of abstraction. The chosen level of abstraction, of course, depends on the level of accuracy targeted in the generated performance estimates.

In general, most frameworks follow the principle advocated in [16, 35, 48] of separating the specification of functionality, communication, implementation and cost. This implies that the application model should only capture functionality and communication requirements, leaving the implementation details and associated cost to be modelled in the architecture model. If the application model includes implementation specific details of the target platform, the application is no longer fully portable and thus constraints the platform onto which the application can be mapped. In general this should be avoided, but in many practical cases where the target platforms are already known, fully or partially, this imposes no problem and in some cases even simplify the mapping process significantly.

At the highest level of abstraction, applications are modelled by e.g. task graphs and only the execution time of the individual tasks is specified. This is the case in ARTS [68] where application models are represented by tasks annotated with an execution time only: i.e. no functional aspects are included.

In most cases, the application model also captures the actual functionality of the

modelled application in order to ensure functional correctness. Several frameworks use a model-of-computation suited for the targeted modelling domain. Kahn process networks, for instance, are used for application modelling in several frameworks targeting stream based applications e.g. [22, 46, 63, 86, 107]. In Koski [46] the use of Kahn process networks is combined with State charts for refinement purposes. In MILAN [11], application models are used to capture the functional behaviour of the application using data flow models-of-computation (ASDF/SDF [60]) only, limiting the applications which can be represented. In Polis [12], Co-design Finite State Machines (CFSM) were used for application modelling, targeted mainly at control-oriented applications seen in e.g. automotive systems.

CASSE [91] is a SystemC TLM based framework and performance estimates are generated through simulations. CASSE also uses a modelling method based on Kahn process networks for modelling applications. Applications are described as process networks in which processes, referred to as tasks, execute concurrently and communicate through the use of point-to-point connections. The individual processes execute sequentially and inter-task communication is synchronized using a proprietary implementation-independent protocol named ITCP.

CHARMED [49] is a framework for automatic multi-objective design space exploration. Applications are represented by high level task graphs. As part of the search process, the tasks of the task graphs are then automatically mapped to the processing elements and communication resources available as specified by the designer in the architecture model. For fitness evaluation, analytical cost estimates are computed and used to evaluate the given solution. The result of the search is a set of non-dominated solutions which describe the best found trade-offs between the targeted design objectives.

Several frameworks use a co-execution approach in which application models are executed generating requests for functions offered by the target architecture; in this way the application model is driving the simulation. This appraoch is taken in e.g. Spade [63, 64, 62], in which the focus is on stream based applications only. Here, applications are specified as Kahn Process Networks as illustrated in figure 2.2. Applications are expressed functionally in C/C++. The application is then annotated with calls from an API supplied by YAPI [24] which implements the Kahn Process Network model of computation. This instrumentation implies that when the C/C++ application is executed, a trace of high level events is generated. The high level trace can then be converted into low level architecture specific traces using a trace transformation technique in which traces of events are re-written to match the features provided by the platform onto which the application is mapped. The trace event instrumentation implies that the application model must be executed on a processing element which knows how to interpret the specified event.

The work of Spade has been continued in the Artemis framework [86, 88] in

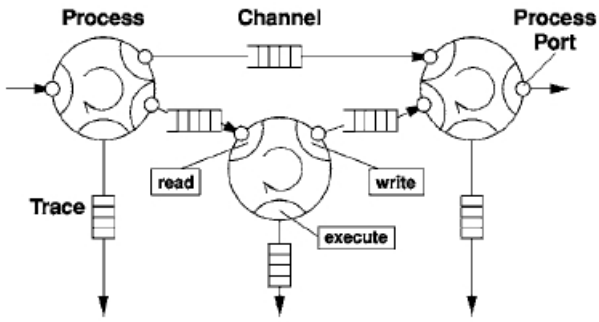


Figure 2.2: Figure from [64], illustrating the use of KPN for application modelling.

which two frameworks for performance estimation at the system level have been developed in parallel in order to investigate different methods. These are the Sesame framework [22, 89, 99, 86, 85] and the Archer projects [107]. Sesame is a framework which is closely related to the work of Spade. In this framework applications are also modelled as Kahn Process Networks which are evaluated during simulation through the generation of event traces which are used as the input to an architecture model.

Archer [107] is also part of the Artemis research project and closely related to Sesame due to the fact that both are derived from the Spade project. However, in Archer, applications specified as Kahn Process Networks are converted into symbolic programs in an attempt to mix the speed of trace driven representations with the accuracy of CDFG based representations which are evaluated on the architecture model during simulation as opposed to the event trace based approach which is taken in Spade.

The Artemis framework also includes Compaan [51] which is a tool capable of converting a subset of sequential Matlab [2] code into Kahn Process networks and the Laura tool [104] capable of converting Kahn Process Networks into synthesizable hardware description language specifications. The Laura tool acts as a backend for the Compaan tool [96].

A recent extension of the work of Spade and Artemis, including their subprojects, is the Daedalus design flow [79, 89, 100]. The Daedalus design flow targets stream based applications mapped to MPSoC based platforms including the modelling of the communication inter-connect fabric (NoC) and includes automated mapping and high level synthesis tools. The Daedalus design flow uses Compaan/Laura for application specification, Sesame for design space ex-

ploration and ESPAM [77, 78] for automated synthesis of high level models into RTL descriptions in hardware description languages.

The SystemCoDesigner [47] presents an approach which supports automated design space exploration and implementation of stream based applications running on systems described as actor oriented behavioural SystemC descriptions using the SystemMoC framework [29]. In this way processes in the application model are described, using an actor, and actors communicate through explicit communication channels. A commercial behavioural synthesis tool is then used to generate RTL descriptions for detailed performance analysis. The designer specifies an architecture model consisting of processing elements and their interconnection as well as the possible mappings and configurations of the individual processing elements. This description is then used as input to an automated design space exploration framework.

The frameworks targeting processor based architectures often specify the application model directly in C/C++ and thus require the existence of compilers for the targeted processors. The compiled applications are then executed on instruction set simulators. Frameworks based on this approach are having great success, both in the industry [5, 4, 97] and in academia [7, 36, 72]. The use of C/C++ to describe applications, however, already places severe constraints on the level of abstraction which can be used to express applications. These frameworks rely on tool chains for building application-specific instruction-set processors (ASIP) including both hardware descriptions and software tools such as debuggers, compilers, assemblers, linkers, etc. A vital element in these tool chains is what is known as architecture description languages, such as LISA [108], nML [30], EXPRESSION [40], etc. These languages make it possible to describe application specific processors at a relative high level of abstraction and then auto-generate software tool chains and even RTL hardware descriptions for input to traditional synthesis flows. In [7], a framework for MPSoC performance estimation is presented which combines a commercially available architecture description language based CAD tool with the MPARM environment [14]. The combination is interesting and detailed information regarding a specific platform can be extracted through simulations.

Frameworks supporting applications to be captured using multiple models-of-computation has been seen as well in e.g. [11, 13, 101]. In these, applications are represented by a number of concurrent execution components which are allowed to communicate through explicit communication channels allowing the individual components to be described using almost arbitrary models-of-computation.

In Metropolis [13] applications are represented as process networks which communicate through ports, which are specified with an interface, declared within the process. Inter-process communication is then handled by connecting the ports of the processes which need to communicate using a medium. The medium

to which a port is connected must implement the interface specified by the port. This allows an explicit separation of the functionality of the processes from communication. The execution of a process is then represented by a sequence of events and a network execution is then an execution of, at most, one event from each process per iteration.

2.3 Architecture Modelling

As in the case of the approach to application modelling, the different approaches to modelling of the actual target architecture are also numerous. The goal of the architecture model is to capture the functionality offered by a specific target architecture and allow a cost to be associated with the application executing on the target architecture. It is the cost of the execution of the application on the specific architecture which differentiates architectures.

In many frameworks, the architecture model is used solely to associate a cost with the execution of the application on the given architecture. The architecture model is thus only focused on capturing temporal aspects, access to resources etc. and not on modelling the actual functionality offered, acting as a cost model only.

In Spade [63], architecture models are constructed using generic blocks provided in a component library. The functional behaviour is captured only in the application model, implying that the architecture model is responsible of modelling the cost of communication and computation only. In order to allow application models to be executed on architecture models, Spade uses the concept of a *trace driven execution unit (TDEU)*. The TDEU accepts the high level events from the application model and translates these into symbolic instructions as input to the actual architecture model. TDEUs are allowed to communicate which each other using a generic protocol. It is a manual task to specify the symbolic instructions and their latency of each TDEU. If multiple application tasks are mapped to the same TDEU, traces need to be scheduled which is done as a round-robin scheduling by default. However, means for providing custom schedulers are provided as well.

The architecture is modelled in Sesame [22] using a library of generic components which can be parameterized. Architecture models do not include any modelling of functional aspects; only the cost of communication and computation is captured because the functionality of an application is modelled in the application model. Architecture components are modelled using Pearl - a discrete-event simulation language [76] - or SystemC. Sesame has extended the work of Spade and uses the concept of virtual processors, instead of the TDEU of Spade, to execute the events generated from the application model. Events from the application model are kept at a very high level of abstraction and a

fundamental concept is that the application models need not to be changed during evaluation of the application on different architecture models. Instead the virtual processor is refined to reflect the model of the architecture so that they are aligned. In practice this is done through the use of a *trace transformation* technique in which the high level event generated by the application model is transformed into a sequence of low level events fed into the architecture model. The trace transformations are specified using Synchronous Data Flow or Integer Controlled Data Flow specifications [89]. The benefit of this approach is that the application model can be specified once and for all but as the architecture model is being described at a lower level of abstraction, specification of the trace transformations must also take place. Essentially, the implementation independence obtained at the application model simply moves the implementation specific trace specification to the mapping layer.

In [101], a framework which also supports the modelling of functionality in the architecture model is described. In this way, different functional implementation specific details can be included in addition to the modelling of cost.

In many cases the focus is on processor based architectures for which high level models of processing elements and abstract operating systems are used. This is the approach taken in ARTS [68], and it allows for very fast performance estimates to be generated. However, it is of course obvious that the accuracy will be rather rough but, nevertheless, it might prove very useful in the early stages of a system level design flow which targets processor based platforms. A more recent approach, very similar to the one taken in ARTS for modelling the target architecture, is presented in [19]. Here, again, high level models of processors and abstract operating systems are also used to compose models of an abstract target architecture. The abstract operating systems are simply modelled as different types of scheduling policies in the case where multiple tasks are mapped to the same processor.

Newer trends for processor-centric architecture modelling, which are referred to as virtual platforms, are also seen. Quite often these models are based on emulation [74] or native execution, possibly with some sort of timing annotation [3, 6], allowing very fast execution times and near-to real-time execution of applications on the modelled architecture.

At lower levels of abstraction, the processor-centric approaches often rely on cycle accurate instruction set simulators which, unfortunately, impacts simulation performance but produce very accurate results. In HySim [36] the performance penalty is limited to some extent and at the same time provides reasonably accurate estimates based on fast Instruction Set Simulators. HySim employs a technique in which the application model is being profiled during the performance simulation and the result is stored in a database. If the part of the application being executed has already been profiled, the profiled result is taken from the data base instead of doing the actual simulation of the block on the in-

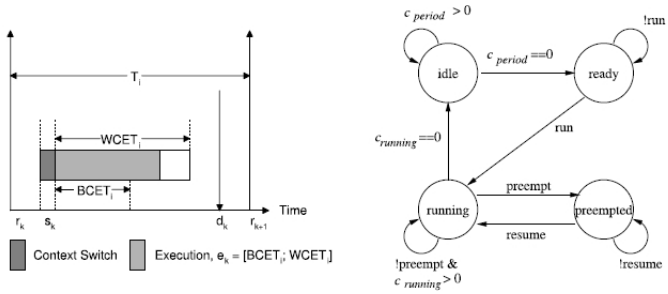


Figure 2.3: ARTS high level representation of tasks, represented by an execution time only, and the possible states of the abstract processing elements used. Figure from [68].

struction set simulator. This boosts the simulation speed significantly; however, it also gives rise to problems when modelling dynamically changing elements which impacts the execution time of a block, e.g. caches etc.

Some frameworks use the notion of services to represent the functionality offered by an architecture model originating from the Polis [12] and the later commercial tool from Cadence, VCC [69] and the joint Philips/Cadence tool COSY [15]. In Metropolis [13], a service is simply a method, representing some functionality, and the efficiency is captured by decomposing each service into a sequence of events. Each event is then annotated with a cost allowing the efficiency of the architecture to be evaluated. Like the application model, the architecture model is composed of processes and media. In this way a decomposition of each service into events is obtained. Quantity managers are then used to model the cost of an event. If an event has an associated cost, the event needs to request the cost from the quantity manger. If the request can be fulfilled, the event can be allowed to execute - if not, the event must wait until the cost becomes available. In this way not only costs but also mutually exclusive access to shared resources is handled by the quantity managers. The architectural model, however, does not itself implement any functionality but is used solely to provide a cost measure.

In [101], services are allowed to represent any piece of functionality offered by both software and hardware components. Furthermore, services are allowed to be composed of multiple sub-services. Costs are associated with services and can be specified statically or computed dynamically at runtime. The cost of a service can be broken down into the cost of sub-services and, in this way, the different cost contributors can be easily identified.

MILAN [11] provides a framework in which a number of readily used simula-

tors can be integrated into a unified environment. For architecture modelling, resource models are used to represent the hardware components of the target architecture. Resource models are specified as meta-models with a specific set of parameters. A hierarchical structure is used which allows inheritance. In this way resource models can inherit shared parameters from models higher in the hierarchy.

In CASSE [91], target architectures are modelled using abstract components much similar to ARTS. Support for processing elements including arbiters supporting multiple tasks mapped to the same processing element, storage elements and communication networks are provided. A proprietary protocol named ICCP is used to handle communication between the components of the target architecture model. The ICCP is an abstract protocol capable of capturing a finite set of real protocols. It is possible to include custom component descriptions in SystemC however; it then requires support for the ICCP protocol to be added.

2.4 Mapping

The partitioning and mapping of applications to a particular target architecture is not a trivial task. This challenge is not addressed here, however, due to the nature and importance of this task; it is interesting to note how the mapping and partitioning process is captured in different system level performance estimation frameworks.

The partitioning and mapping process is not always explicitly represented. In many frameworks, the mapping of an application to the target architecture is of great importance performance wise. Frameworks which expose the mapping step to the user explicitly often allow fewer changes to the system model during design space exploration.

In general, in order to facilitate an easier mapping process the application models should:

- Expose task level parallelism
- Have explicit communication requirements.
- Be specified functionally only.

The exposure of the mapping step to the designer allows the designer to investigate the effect of different mappings. Suppose that the individual tasks of an application requires functionality offered by several different components of the target architecture; the mapping to the best suited components in terms of a given cost metric, then, is dependent on several factors, some of which are runtime specific.

In order to evaluate a given application on a given architecture in Metropolis [13], an explicit mapping is performed. The mapping relates the events of the functional model with the events of the architectural model. This is done by specifying which events of the two models that must be synchronized. In this way the actual execution of a given application model on a given architecture can be modelled.

MILAN [11] uses constraint models to specify different constraints which a given model must adhere to. Together with resource and application models, these are used for composing the structural system model and in this sense the constraint models are also used to specify valid mappings.

2.5 Performance Estimation

Performance estimation is the process of associating one or more quantitative cost measures with the system and/or the individual parts of the system in order to asses the quality of a given system. The cost metric of a given system will then be the differentiating factor of systems which offer the same functionality. Cost metrics can, of course, be associated with systems that are physically present. However, in order to explore a large set of systems, it is of vital importance that cost metrics can be associated with systems which are not necessarily physically realized, and thus, allows estimates of the performance of a given system to be produced, thereby permitting a cost to be associated with the system. In this way designers can be allowed to choose the best suited system from well defined criteria.

In most cases the objective space is multi-dimensional, and hence the process of finding the best suited system can be categorized as a multi-objective optimization problem. However, in order to start trying to solve this problem, which is hard enough in it self, reliable performance estimates of the system must be available.

Performance estimation of a system can be performed both analytically and/or through simulations. Analytical methods are most often seen in high level performance estimation methods, whereas simulation based methods are found at all levels of abstraction. The big benefit of the formal analytical methods is that these methods are very often exact under the given assumptions, whereas simulation based methods need not find worst case execution paths.

Important general cost metrics in the domain of embedded systems are the execution time, the power consumption, the total cost, the silicon area, etc. of a given application on a given target architecture. Typically, however, there are application specific cost measures of vital importance as well; thus, a framework

providing the possibility of estimating performance should be very flexible and support the possibility of having user defined cost models.

For higher levels of performance estimation, frameworks for the development of virtual platforms exist. Although, the main focus is on providing means for allowing early software development to take place, these frameworks are also used for rough performance estimation. In general these frameworks are focusing on providing a virtual platform for software developers to use in the early design stages before a physical hardware platform has been realized. Mostly, these platforms require that applications are already present as source code in e.g. C/C++. Several of these frameworks are rooted in the instruction set simulator community and are now being extended to be used in complete system level designs. The approaches taken in these frameworks are often a combination of emulation, interpretation and compiled simulation in order to achieve fast simulation speeds. These approaches are very useful for early software development which needs not to wait for the initial hardware bring-up. However, despite being close to functionally equivalent to the target architecture, many of the approaches which obtain fast execution times produce only very rough estimates of the actual system performance.

Some frameworks [27, 36], suggest that costs of the execution of a specific block should be computed once only and then retrieved when executed again which will speed up the performance estimation process. In [102] a framework for performance estimation at the system level is presented based on the profiling of a high level system model described using SystemC. Data transfers are recorded during the general profiling step and the collected trace is used to construct a graph representing the execution order of the system. The execution order graph is then used to generate what is called an architecture level dependency graph, taking into account the actual execution on the target architecture, which is specified by the designer as an architecture model. The graph representing the execution order of the system is generated only once, whereas different architecture level dependency graphs are constructed based on the given target architecture. Using the information about the architectural level execution order, it is possible to generate performance estimates of the system. Performance estimates are calculated analytically based on information from an IP data base which is assumed to be available. Only rough estimates can be generated using this approach; however, this can be done relatively fast as shown in [102].

Trace-driven simulation methods for capturing workload and/or behaviour are also seen. This is an approach which is inspired by the memory modelling community in which trace-driven simulations of memory access have been investigated thoroughly [103]. From the simulation of an application model, a trace is generated and then fed into the architecture model in order to associate a quantitative cost measure with the execution of the application. However, trace-driven simulation methods often experience problems when modelling dy-

namic effects such as caches, interrupts etc. In [57] traces are also generated from application models, but, in this case the objective is to statically analyse the communication requirements of the application in order to design a communication architecture.

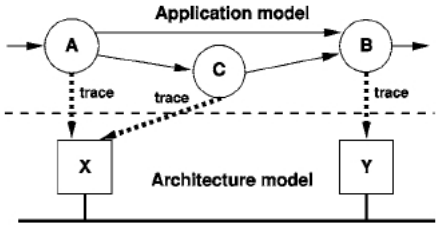


Figure 2.4: Figure from [64], illustrating the principle of trace-driven co-simulation.

A number of frameworks uses a variation of the trace-driven simulation referred to as trace-driven co-simulation[22, 63, 107], in which traces are generated and evaluated dynamically, as illustrated in figure 2.4. Dynamic requests are generated to functions offered by the architecture by the application models. In this way allowing e.g. implementation specific cost to be modelled, while still being able to capture dynamic effects.

In [58] a hybrid trace-driven approach for system level performance estimation based on a combination of simulation and static analysis of traces is presented. The objective of the framework is to assess system level performance in communication architecture design; thus, the main emphases is on capturing the communication requirements of the application and then design an optimal communication architecture through the application of iterative refinements. Abstract traces are generated through co-simulation which is then used to estimate the performance of particular communication architectures through an analytical transformation of the abstract trace; the co-simulation is only performed once.

2.6 Summary

In this chapter an introduction to the field of system level modelling and performance estimation was given. A number of frameworks and methods which can be used partly, or fully, for performance estimation at the system level were presented.

As can be seen from this overview, the area of system level modelling and performance estimation is a true multi disciplinary field which have been the focus of much research for more than a decade under the name system level design and even longer under different names or interpretations of the name. This chapter categorized related work within the categories abstraction level, application modelling, architecture modelling, mapping and performance estimation because these are the main ingredients in the system level modelling and performance estimation framework presented in this thesis.

Only a few of the frameworks discussed can be compared directly to the framework presented in this thesis within all categories. However, many frameworks share similarities within some aspects e.g. with regards to application modelling or platform modelling. However, the frameworks which are most closely related to the one which is presented in this thesis are the Metropolis [13] and Artemis [88], Spade [63] and Sesame [22] frameworks. Table 2.1 summarizes some chosen parameters for a number of the more elaborate frameworks and tools which have been discussed in this chapter in order for these to be comparable. It has been chosen to use parameters which express how systems are represented as either being Y-chart supported or not, if they can represent arbitrary application types, i.e. not being targeted a specific application domain, whether multiple models-of-computation can be used and allowed to co-exist for capturing the behaviour of both the application and the targeted architecture. Finally, parameters such as whether the tool or framework is simulation based, supports formal analysis and automated design space exploration are presented.

Metropolis [13] is one of the most comprehensive and theoretically well-founded frameworks publically available. However, due to a number of limitations especially regarding the practical mapping of applications onto platform, as discussed in [27], the framework is not widely adopted. The vast amount of research and experience, however, is now continued in Metro II [27] which addressed these exact limitations. Spade [63] and related projects are also very comprehensive works which allow performance estimation at the system level. However, these frameworks only target stream based applications limiting the scope of the use to the class of multimedia applications.

As already stated in chapter 1, this thesis presents a framework for system level modelling and performance estimation using a unified modelling approach for capturing both hardware and software parts, supporting a gradual refinement of components and allowing multiple models-of-computation to co-exist and communicate.

	Y-Chart	Arbitrary applications	Heterogeneous MoC	Mixed-abstraction level	Simulation	Formal analysis	Automated DSE
MILAN [11]	No	No	Yes	No	Yes	Yes	Yes
Artemis [88]	Yes	No	No	Yes	Yes	No	No
Spade [63]	Yes	No	No	Yes	Yes	No	No
Sesame [22]	Yes	No	No	Yes	Yes	No	Yes
SystemCoDesigner [47]	Yes	No	No	Yes	Yes	No	Yes
MESH [18]	No	Yes	No	No	Yes	No	No
COSY [15]	Yes	Yes	No	No	Yes	No	No
Polis [12]	Yes	No	No	Yes	Yes	Yes	No
Metropolis [13]	Yes	Yes	Yes	Yes	Yes	Yes	Yes
MESCAL [72]	Yes	Yes	Yes	(Yes)	Yes	No	Yes
Ptolemy II [28]	No	Yes	Yes	Yes	Yes	No	No
CHARMED [49]	Yes	(Yes)	No	(No)	No	Yes	Yes
CASSE [91]	Yes	No	No	(Yes)	Yes	No	No
This work	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**No**	(**Yes**)

Table 2.1: Comparison of some of the more comprehensive frameworks and tools discussed in this chapter.

On the other hand, the framework presented here, lacks the support for formal analysis which is found in Metropolis and currently only automatic design space exploration is provided at a level which is mostly meant as proof-of-concept.

Chapter 3

Service Models

This chapter introduces the service model which is the fundamental modelling component of the system level modelling and performance estimation framework presented in this thesis. Service models are used for modelling the individual components of which an embedded system is composed using a unified modelling approach capable of capturing both hardware and software elements.

The service model can be viewed as a meta-model capable of capturing the behaviour of the components of which the application and the target architecture are composed at arbitrary levels of abstraction. The meta-model allows different models-of-computation to co-exist and provides semantics for inter-model communication across abstraction levels and provides a clean and simple solution for separating the specification of functionality, cost, communication and implementation following the principles advocated in [48].

In the following, the basic properties of service models will first be introduced. The means for separating the modelling of functionality, implementation, communication and cost will be presented and the composition of service models will be explained. Finally, it will also be discussed how the existence of multiple abstraction levels are supported within the same model instance and how abstraction level refinement can be performed.

3.1 Service Models

A service model captures the functional behaviour of a component through the notion of services. Services are used to represent the functionality offered by a given component without making any assumptions about the actual implementation. Concrete examples of services are functions of a software library, arithmetic operations offered by a hardware functional unit or the instructions offered by a processor depending on the level of abstraction used to model the component.

The functionality offered by a service model, represented as services, can be requested by other service models through requests to the offered services. The detailed operation of a service model, i.e. the implementation of the services offered, is hidden to other models. In that sense a service model can be viewed as a black box component and it is only the services offered which are visible and not their implementation. Service models can be described at multiple levels of abstraction and need not be described at the same level in order to communicate. The clear separation of functionality and implementation through the concept of services implies that there is no distinction between hardware or software components seen from the point of view of the modelling framework. It is the cost associated with each service which eventually dictates whether a component is modelling a hardware or software component.

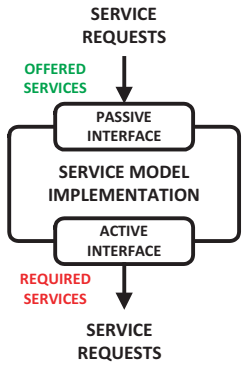


Figure 3.1: The service model basics.

A service model is composed of one or more *service model interfaces* and a *service model implementation* as illustrated in figure 3.1. Interfaces are used to connect models, allowing models to communicate through the exchange of service requests. In this way, there is a clear separation of how behaviour and communication of a component is described. The service model interfaces provide a uniform way of accessing the services offered as well as enabling service models to communicate and facilitate structural composition by specifying the

sets of services which are offered and are required by the model. It is the active interfaces which specify required services and the passive interfaces which specify the offered services. In this way, structural composition is dictated by the interfaces each service model implements and subsequently by the services offered and required. Only models fulfilling the required-provided service constraints can be connected together. Whereas it is the interfaces which dictate the compositional rules, it is the service model implementation which is responsible for specifying the actual behaviour of the service model by implementing the functionality of each of the services offered and possibly associate a cost with these.

The use of services allows a decoupling of the functionality and of the implementation of a component. In this fashion, several service models can offer the same service, implemented differently, however, and thus having different costs associated. In this way different implementations can be compared and evaluated based on a specific, preferred cost metric.

Inter-model communication is handled through service requests which have the benefit of allowing the initiator model to request one of the services specified, as required by the model, without knowing any details about the model which provides the required service or how it is implemented. When the service is requested, the service model providing the required service will execute the requested service according to the specification of the model. When the service has been executed, the initiator model is notified that the execution of the service requested is done.

3.2 Service Model Interfaces

In order to facilitate structural composition of models and to allow communication between models, possibly described at different levels of abstraction, interfaces are used. The interfaces directly impose the compositional rules which specify how models can be connected by allowing only interfaces fulfilling the required-provided service constraints to be connected.

The use of interfaces also implies that multiple implementations of a service model can be constructed and be seamlessly interchanged allowing different implementations to be investigated and described at different levels of abstraction, constrained only by the requirement that the service model implementations considered must implement the same interfaces.

Two types of interfaces are defined:

- The *Passive service model interface* specifies a set of services that the model which implements the interfaces offers to other models. The passive

interface also includes structural elements allowing the interface to be connected to an active interface as well as provide means for requesting the services offered.

- The *Active service model interface* specifies a set of services which are required to be available for the model which implements the interface. The set of required services becomes available for the model which implements the active interface, when the active interface is connected to the passive interface of a service model which offers a set of services in which the required set of services is a subset.

Active service model interfaces can only be connected to passive service model interfaces in which the set of services required by the active service model interface is a subset of the services offered by the passive service model interface. In essence, the compositional rules, which specify which active service model interfaces can be connected to which passive service model interfaces, is dictated by the services required and services offered by the two connecting interfaces.

3.3 Service Model Implementations

In order to capture the behaviour of a service model, the service model implementation must be defined. It is the service model implementation which captures the actual behaviour of a service, possibly taking implementation specific details into consideration. A service model must provide an implementation of all services offered by the passive interfaces implemented and optionally specify the latency, resource requirements and cost of each service. There are no restrictions on how the implementation of a service should be made. The abstract representation of the functionality, using services, implies that there is no immediate distinction between the representations of hardware or software components. Whether a service model represents a hardware or software component is determined solely by its implementation and, eventually, its cost and thus a very elegant unified modelling approach can be achieved.

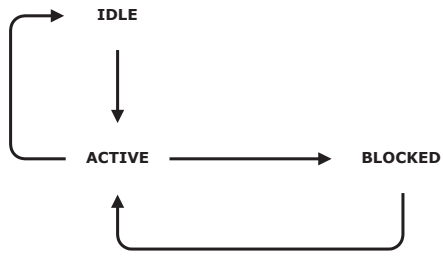


Figure 3.2: The possible states of a process of a service model.

Service models use concurrently executing processes as the general execution semantics. The service model implementation can contain one or more processes which each have the possibility of executing concurrently. A process executes sequentially and inter-process communication within a service model is done through events communicated via channels in which the order of events is preserved. All inter-model communication between processes residing in different service models is done using service requests via the service model interfaces defined by the model in which the processes reside.

A process can be in one of three states: Idle, active or blocked as shown in figure 3.2 which also shows the valid transitions between the possible states. If a process is idle, it indicates that it is inactive but ready for execution upon activation. If the process is active, it is currently executing. If a process is blocked, it is currently waiting for a condition to become true and will not resume execution until this condition has been fulfilled.

In order to overcome the problem of finding a single golden model-of-computation for capturing all parts of an embedded system, the service model concept supports the existence of multiple different models-of-computation within the same model instance. Figure 3.3 shows an example of a system composed of service models, each described by a different model-of-computation. The service model concept allows these to co-exist and communicate through well defined communication semantics in the form of service requests being exchanged via active-passive interface connections.

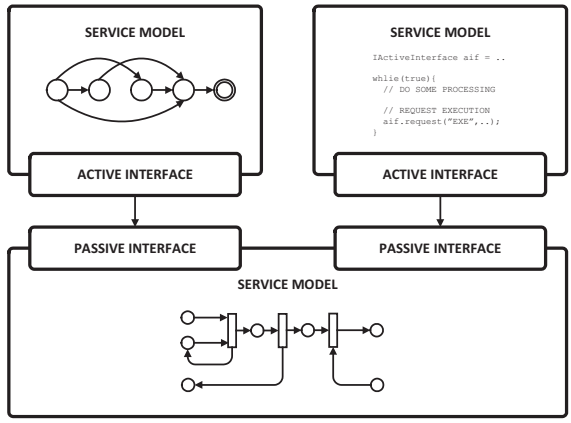


Figure 3.3: The service model concept provides support for heterogeneous models-of-computation to co-exist.

Interesting work on supporting multiple different models-of-computation within a single model instance is presented in the theoretically very well founded tagged signal model [59] and in the absent-event approach [43]. Both approaches show

that it is possible to allow models-of-computation, defined within different domains, to be coupled together and allowed to co-exist within the same model instance. In principle, the service model concept does not impose a particular model-of-computation; the individual models can be described using any preferred model-of-computation as long as communication between models is performed using service requests.

Currently, the focus of the framework is the modelling of the discrete elements of an embedded system only, i.e. hardware and software parts which can be represented by untimed, synchronous or discrete time based models-of-computation. Most embedded systems also contain elements of an analogue nature either in the form of analogue electronics or if parts of the environment need to be modelled in order to generate accurate performance estimates. In such cases, the service model approach could most likely be used as well. To be conclusive, this requires more in-depth investigations. The idea, however, is that one would have the analogue element described by a continuous time model-of-computation which would then be evaluated at discrete time instances defined by the arrival of a service request to the model representing an analogue element. Care should be taken in order to obtain correct behaviour, as this would correspond to performing an A/D or D/A conversion depending on the direction of the communication which includes a number of pitfalls which will not be disused here.

In order to synchronize models described using different models-of-computation and ensure a correct execution order, the underlying simulation engine is assumed to be based on a global notion of time which is distributed to all processes, no matter which model-of-computation is used, as opposed to both the tagged signal model and the absent event approach which distributes time to the different processes through events and the special absent event respectively. The drawback of using a global notion of time is that processes cannot independently execute which impacts simulation performance - it is very hard to parallelize such a simulation engine - however great expressiveness can be obtained. The simulation engine also tags all events with a simulation time value so that the event can be related to a particular point of simulation time no matter which model-of-computation was used to describe the process that generated the event. However, this does not mean the individual service models need to use this time tag and it is merely a practical requirement in order to schedule the execution order of the processes of the individual service models. A realization of such a simulation engine will be described in more detail in section 5.2 in which a delta-delay based notion of time is used.

Service models which have processes described using untimed models-of-computation obviously have no notion of time and perform computation and communication in zero time. This implies that a process of a service model described using an untimed model-of-computation is activated on the request of services offered by the model only.

Service models having processes described using synchronous models-of-computation do not use an explicit notion of time; instead a notion of time slots are used. In order for such models-of-computation to be used, the service model using a model-of-computation within this domain must specify the frequency of how often the processes of the model should be allowed to execute. The simulation engine will then ensure that the processes are evaluated at the specified frequency, in this way implicitly defining the actual time of the current time slot of the model.

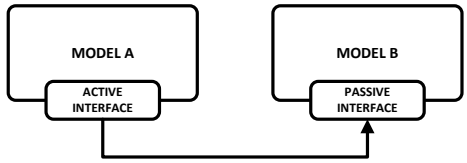
Timed models-of-computation are supported directly by the simulation engine which provides a global notion of time which can be accessed from all processes. In this way, a process can describe behaviours which use timing information directly.

The generality of service models impose only a few restrictions on the model-of-computation used to capture the behaviour of the component being modelled. New models-of-computation can be added freely under the constraint that they *must* implement inter-model communication through the exchange of service requests and they must fit under the general execution semantics defined - i.e. it must be possible to implement the preferred model-of-computation as one or more concurrently executing processes. It is the implementation of the service models which determines their actual behaviour and thus it is the designer of the service model implementation who determines the model-of-computation used.

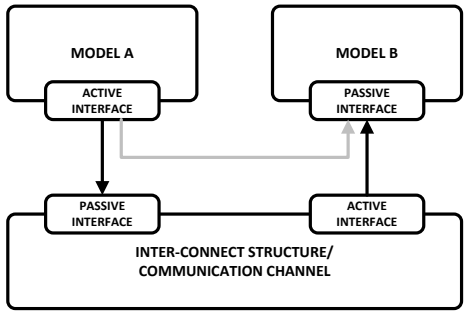
3.4 Service Requests

As mentioned, all inter-model communication is modelled using service requests exchanged via active-passive interface connections. A service request can be viewed as a communication transaction between two components, similar to the concept of transaction level modelling [16]. However, it is the implementation of the two service models which will determine if the service request will be implemented as a function call from e.g. a sequential executing piece of code to a function library or if it will be a bus transfer. Communication refinement is supported in several ways. Service requests can include arbitrary data structures as preferred by the designer of the model. If a communication channel needs to be modelled, an extra service model can be inserted between the two primary communicating service models as illustrated in figure 3.4. The extra service model inserted will be transparent to the two communicating service models. In this way e.g. simple properties such as the reliability of the communication channel can be modelled. More elaborate communication inter-connects such as buses; network-on-chips etc. can also be modelled.

A service request specifies the requested service, a list of arguments (which can



(a) Model A is the active model, initiating communication with model B.



(b) The communication medium is now modelled explicitly using a seperate service model

Figure 3.4: Inter-model communication.

be empty) and a unique request number used to identify the service request, e.g. in order to annotate it with a cost. The argument list can be used to provide input arguments to the implementation of a service, e.g. to allow the modelling of dynamic dependencies or arithmetic operations on actual data values. Depending on the implementation of the service model, an arbitrary number of service requests can be processed in parallel, e.g. modelling operating system schedulers, pipelines, VLIW, SIMD, and super scalar architectures.

A service request can be requested as either blocking or non-blocking. It is the designer of a model who determines whether a service request is requested as a blocking or non-blocking request. The request of a blocking service request implies that the process of the source model which requested the service request is put into its blocked state until it has been executed, indicated by the destination model, as illustrated in figure 3.5. A non-blocking service request, on the other hand, will be requested and the process of the source model, which requested the service request, will carry on - not waiting for the execution of the request to finish.

A number of events are associated with a service request in order to notify the requester and receiver model of different phases of the lifetime of the service request. The lifetime and corresponding events of a service request is as

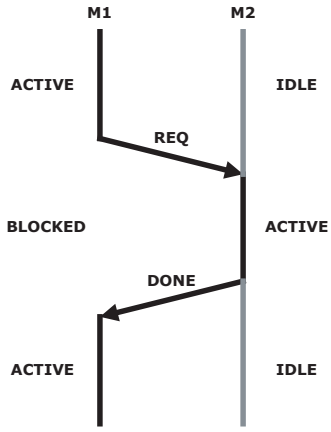


Figure 3.5: A process of model *M1* requests a blocking service request from *M2*. *M1* is blocked until the completion of the request.

follows:

1. The service request is being requested, indicated by a **service request requested event**.
2. The service request is being accepted for processing of the model by which it is requested, indicated by a **service request accepted event**.
3. The service request might be blocked, indicated by a **service request blocked event**
4. The service request has been executed, indicated by a **service request done event**.

When a service request is being requested at a model interface, the receiver model has the possibility of receiving a notification in order to change its status to active. The requesting service model will similarly have the possibility of being notified when the request is accepted for processing in the receiver model, i.e. prior to the actual execution of the service request, as well as when it has finished executing the service request. During the evaluation of a service request, the request itself can become blocked due to one or more requirements not being fulfilled, e.g. due to mutually exclusive access to resources, missing availability of data operands, etc. When a service request is being blocked during evaluation, the source model is notified in order to allow it to take appropriate actions, if any. However, the author of the requesting service model need not be interested in receiving these notifications and, hence, the model is allowed to ignore these. In this way it is the designer who chooses event sensitivity for the individual processes of the service model implementation.

In order to handle multiple simultaneous service requests, the designer of a service model must incorporate a desired arbitration scheme. The arbitration scheme may be an integrated part of the model or it may be a separate service model itself. The latter is often advantageous if different arbitration schemes are to be investigated.

3.4.1 Composition

In order to tackle complexity, the service model concept supports both hierarchy and abstraction level refinement. In this case, the term abstraction level refinement covers the process of going from a high level of abstraction to a lower level through gradual refinements of a given component, in this way replacing one component with a more detailed version as illustrated in figure 3.6.

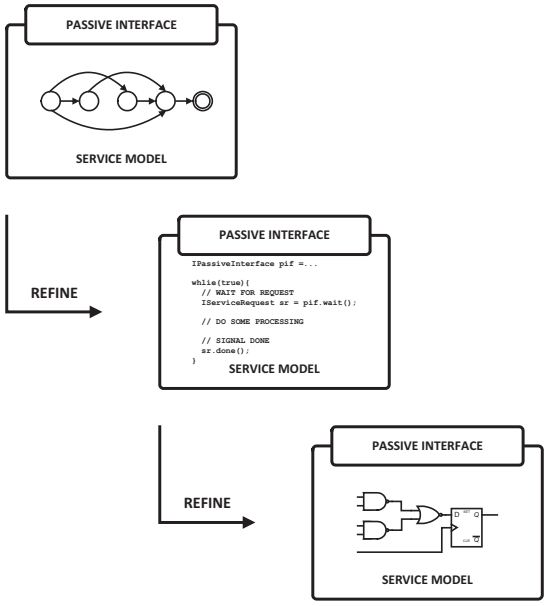


Figure 3.6: Abstraction level refinement.

This type of refinement is supported quite easily by the service model concept because of the fundamental property of the service model in which the functionality offered by a model is separated from the implementation. Two service models implementing the same set of interfaces, and thus offering the same set of services, can be freely interchanged even though they differ in the level of detail used to model the functionality offered.

Furthermore, service models can be constructed hierarchically in order to investigate different implementations of a specific subpart of the model or in order to hide model complexity. One service model can be composed of several subservice models. However, it will then be only the interfaces implemented by the topmost model in the hierarchy which dictate which services are offered to other models. The hierarchical properties, combined with the use of interfaces, imply that designers who are using a model need only know the details of the interfaces implemented by the model and need not be concerned about the implementation details at lower levels in the model hierarchy.

3.5 Summary

A service model can be viewed as a black-box component. The behaviour of a service model is determined by the services requested via its active interfaces. The use of interfaces and service requests implies that there are no restrictions on the service model implementation which is the part of the service model that actually determines the behaviour.

The simplicity of the unified model for capturing both hardware and software is one of the strongest properties of the service model concept. There is a clear separation of the specification of functionality and implementation, and designers need not think about how a component will be implemented, nor if it will be implemented in hardware or software until a desired point in the refinement process. This allows a broader search of the design space because the decision of how to implement a given component will be based on the system model which has the best performance in terms of a well-defined cost-metric based on quantitative performance estimates generated through the use of the presented simulation based system level performance estimation framework.

Furthermore, the service model concept allows a gradual refinement of components as desired by the designer. The goal of the gradual refinement of the components is to reach a level of abstraction from which the actual synthesis of the implementation of the system can take place.

Chapter 4

A Framework for System Level Modelling and Performance Estimation

In this chapter the system level modelling and performance estimation framework which is one of the main contributions of the work carried out in the course of this project will be presented. The chapter presents the major elements of the framework and describes how performance estimation at the system level can be carried out.

The framework presented in this thesis does not strictly enforce the use of a specific design methodology and thus can be used in several ways. However, as already mentioned, the framework is related to the Y-chart approach [12, 50] and leverage principles of the Platform Based Design (PBD) paradigm [48]. As a consequence, a separate specification of the application (functionality) and the target architecture (implementation) is used in the framework in the form of an application model and platform model respectively. In this way favouring design methodologies based on these principles.

Abstraction level refinement is performed independently for the application and platform model as illustrated in figure 4.1. Thus, the possible abstraction level of the system model is a 2-dimensional plane spanned by the abstraction levels of the application and platform model respectively. Different paths can be taken through the abstraction level space of the system model as shown, e.g. **A** and **B** in the right part of figure 4.1. The framework, however, imposes no

requirements of how the refinement process should be performed. In most cases, a top-down approach is preferable but not mandatory. Using the framework in a top-down approach allows an iterative refinement of models, lowering the level of abstraction used to describe the system. The iterative approach ensures that at each iteration level, the refined model can be verified against the previous level, making the task of verification easier. At some point the abstraction level of both the application model and platform model will reach a level suitable for implementation as illustrated in the point referred to as implementation model in figure 4.1.

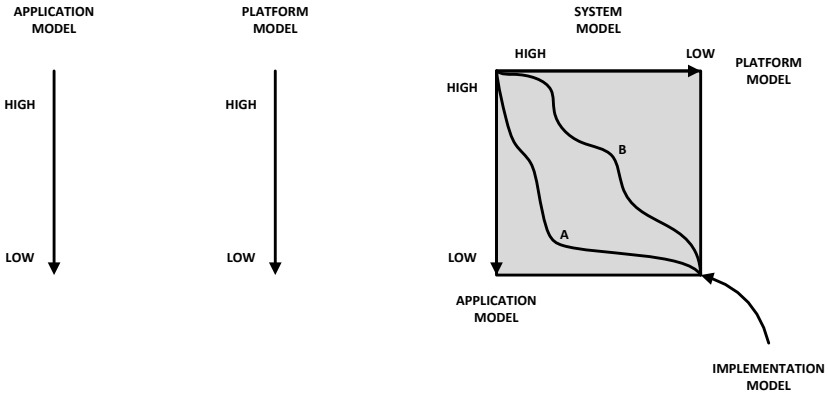


Figure 4.1: Abstraction level refinement. The application model and the platform model can be refined individually, thus the resulting abstraction level of the system model is a 2-dimensional plane spanned by the application and platform model abstraction levels respectively.

In the following, it will first be discussed how the functional behaviour of applications are captured in the form of an application model followed by a description of how models of a target architecture are constructed and modelled in a platform model. Finally, the focus will be on the complete system model which relates the application model to the platform model through an explicit mapping step and how performance estimation can be carried out through simulations of a system model.

4.1 Application Modelling

In the framework, applications are represented by *application models* which are composed of an arbitrary number of parallelly executable components, referred to as tasks. Application models are used to capture the functional behaviour and communication requirements of the application only. The goal of an application

model is to expose as much inherent task level parallelism as possible, as well as the inter-task communication requirements, in order to ease the mapping process.

In the general case, a strict separation of the functional behaviour, communication requirements and implementation of an application must be applied in order for the application to be platform independent. Thus, no assumptions on how the application is implemented should be made. However, in some cases it might actually be desirable to include platform specific information in the application model, e.g. in the case where an existing platform is being modified and, thus, support for including implementation specific details in the application model is provided. However, it will reduce the number of platforms onto which the application model can be mapped.

As already mentioned, the tasks of an application model communicate through the exchange of service requests, as described in section 3.4, making communication explicit and implementation independent, following the concepts of which the framework is founded of separating functionality from implementation. Inter-task communication can occur directly between tasks or, if preferred, via abstract buffers. Abstract buffers are also modelled using service models and can be bounded or unbounded. There are no restrictions on the type of buffers which are supported - the type of buffers used is determined by the designer of the application model. Thereby, a separation of the functional behaviour of a task and the communication requirements is obtained. This makes communication explicit which makes the later mapping of the task easier and, more importantly, it makes the application model independent of the underlying implementation.

Figure 4.2: A simple application consisting of three tasks.

Figure 4.2 shows a simple application consisting of three tasks. In an application model each task is represented by a service model, as shown in figure 4.3, each having a single process associated. In this way tasks have a process-like behaviour during simulation. This implies that tasks are modelled as executing concurrently and that the individual task executes sequentially until it makes a blocking service request through one of the active interfaces implemented by the service model representing the task.

Application models can be executed and used for verifying the functional behaviour of the model and in this way can be used as an executable specification of the targeted functionality; however, at this level of abstraction there is no notion of time, resources or other quantitative performance estimates. In order to

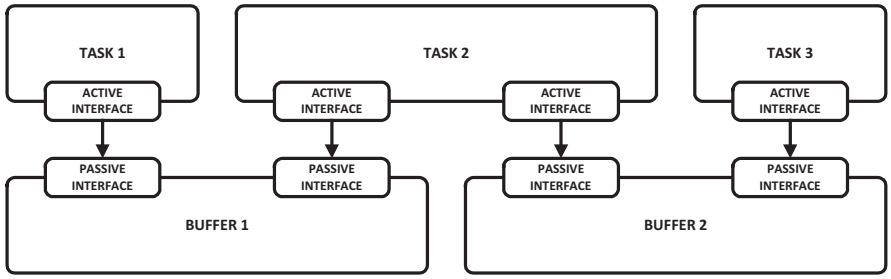


Figure 4.3: The corresponding application model for the application shown in figure 4.2 with tasks and implicit buffers represented as service models.

obtain these, the tasks and buffers of the application model must be mapped to the components of a platform model. When the tasks of an application model are mapped to the processing elements of a platform model, the tasks, when executed, can request the services offered by the processing elements, modelling the execution of a particular functionality or set of functionalities. The services required by an application model must be offered by the platform models in order to be valid candidates for execution of the given application model.

The service model concept provides great flexibility and several different modeling approaches are supported for capturing applications. In general, the concept allows a dynamic co-execution of application and platform components through the unified representation of hardware and software elements. Services offered by other components are requestable through the concept of service requests. This implies that service requests can either be generated dynamically during simulation at one extreme and, at the other, simply be a recorded trace which is being played back.

In the next section, it will be described how quantitative performance estimates can be associated with the execution of an application model using a model of the target architecture represented by a platform model.

4.2 Architecture Modelling

In order to capture the behaviour of a given target architecture platform, models are used. In contrast to the application models, the goal of the platform model is to capture a specific implementation of the functionality offered by the target architecture.

A platform model must offer *all* the services required by the application models, which are mapped to the platform, in order for the platform to be valid. Several

platform models can offer the same set of services, representing the same functionality, through different implementations. The differentiating factor, however, will then be the cost associated with the execution of the applications on each of the candidate platforms.

The platform model is used for two purposes. Firstly, the platform model allows costs to be associated with the specific system realization, allowing designers to associate a cost with the execution of an application on the specific platform. The individual service models of a platform model associate a cost with each service offered determined by the designer of the model. Secondly, implementation specific functionality can be modelled in the platform model so that e.g. the computation of a given value is done in accordance with the actual implementation.

In this way, it is the service models of the platform model which allow designers to associate quantitative cost with the execution of an application model. Without a platform model, the service models of an application model simply execute in zero time and have no cost associated. When the tasks of an application model are mapped to the processing elements of the platform model, it becomes possible to capture resource requirements, quantitative costs and associate a time measure with the execution of the application model on the target platform. In this way, it is the platform model which brings a notation of time into the simulations and thus orchestrates the execution of a given application model with respect to the timing and resource requirements defined by the individual parts of the application model executing on the platform model.

The elements of a platform model can be both hardware and software and depending on the level of abstraction used, the decision as to which components are implemented in hardware or software need not be decided until a late stage in the design process. Through a refinement of the platform model, a level at which the co-design problem of choosing which components are implemented in software and hardware can eventually be addressed. However, in most practical cases, in order to associate a reliable cost with the functionality offered by a given component, considerations of how the functionality offered is implemented should be made.

The platform model of a given target architecture is implemented as a service model having one or more passive service model interfaces, as illustrated in figure 4.4. Platform models are composed of an arbitrary number of service models, each modelling a component of the target architecture and, so, form a hierarchical model. The compositional properties of service models even allow multiple platform models to be merged into a single platform model. In this way, a modular approach can be taken in which sub-blocks of the target architecture is modelled and explored individually if preferred.

The resulting set of services offered by a platform model is dictated by the

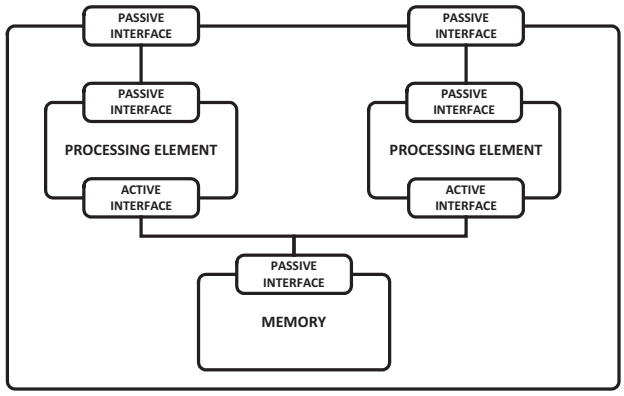


Figure 4.4: Illustration of a simple platform model consisting of two processing elements both connected to a block of shared memory.

composition of internal service models. From the tasks of the application models, which are mapped to the platform, the services of the platform model are accessible through the passive service model interfaces, allowing the task to request the services offered during simulation, modelling the execution of the task.

The platform model also specifies how the service models of which it is composed are inter-connected, thereby specifying the communication possibilities of the models. There are no restrictions on how inter-component communication is modelled, nor on the level of abstraction that is used, implying that, in principle, all types of inter-component communication methods are supported. In this way, platform models can represent arbitrary target architectures.

4.3 System Modelling

In order to capture the complete system, a system model, constructed by mapping the service models of one or more application models onto the service models of a platform model, is used.

The application model is a functional model of the application and does not make any assumptions on the implementation. Thus the application model does not in itself address the hardware/software co-design problem; this important problem is not decidable until the application model is being mapped to a platform model. The separate specification of the application and platform model, following the principles of the y-chart methodology, implies that one application model described at a particular level of abstraction can be mapped to

an arbitrary number of platform models, offering the required services, without changing the application model. In this way allowing different platforms to be evaluated and compared or even the same platform captured at different levels of abstraction.

Quantitative costs can be associated with the services offered by a platform model, and through a co-execution of the application model and the platform model, the total cost of executing the application model on the specific platform model instance can be calculated. In this way it is the application model which is *driving* the simulation by requesting services offered by the platform model.

The tasks of an application model can be categorized as being either *functional*, *mixed* or *implementation* tasks when used in the composition of a system model as illustrated in figure 4.5. The categories do not constrain the level of abstraction used to describe the tasks but are used solely to express how the functionality of the task is being modelled within the system model. In general, the obtainable accuracy is highest using tasks belonging to the implementation category, however, this need not to be the case.

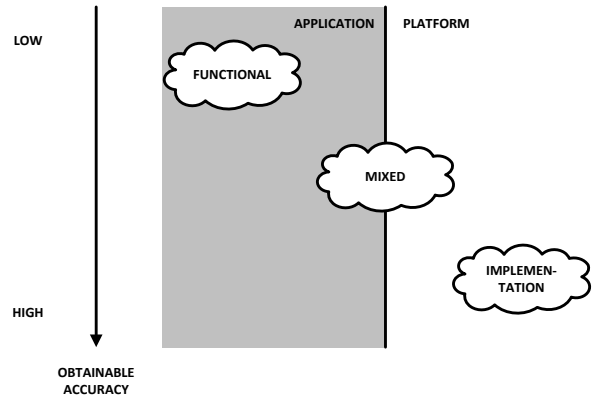


Figure 4.5: Illustration of the tasks categories. In general, the obtainable accuracy is highest using tasks belonging to the implementation category, however, this need not to be the case.

A functional task is modelled solely within the application model and has no costs associated. Such task types are used for behavioural purposes only.

A mixed task is a task which is mapped to a processing element of the platform model in which only part of the functionality is modelled in the platform model, leaving the remaining part to be modelled in the application model. Cost and resource requirements can be included as well, depending on the abstraction level

used to describe the processing element of the platform model. An example of a mixed task could be when a task is implementing loop control in the application model directly, and only part of the functionality and/or the cost of the loop body through requests to services offered by the service model of the processing element in the platform model onto which the task is mapped.

The use of mixed tasks is particularly useful in the early stages of the design process where rough models of the platform might be constructed. In such a scenario, fast estimations of the effect of adding redundant hardware support for specific operations, or even rough estimates of the effect of using multi processor systems, the effect of buffer sizes etc. can be explored. This scenario might be refined to a level where cycle accurate and bit true models are described but still leaving the control flow of the application to be handled in the application model and modelling only the cost of control operations in the platform model.

An implementation task is a mapped task which is represented solely by a set of requests to the services offered by the processing element of the platform model onto which it is mapped. In this case, the complete functionality is modelled in the platform model. Consequently, the complete task is represented by a service request image, directly equivalent to a binary application image for a processor, and the application model is not evaluated during simulations which, compared to models described at the same level of abstraction, results in a speed-up in simulation time. A second advantage of the support for implementation tasks is that a platform model described at this level of abstraction can also be used for performance estimation of compiler technologies.

It is important to notice that tasks belonging to the three categories can be mixed freely, providing great expressiveness and flexibility. Thus, depending on the level of abstraction used to describe the service models of a platform model, the execution of a requested service can model the actual functionality of the target architecture. Furthermore, the required resources and their cost in terms of e.g. latency or power can be included. Thereby, the functionality of the application model mapped to the platform model is modelled according to the actual implementation, including e.g. the correct bit widths and availability of resources, simply by refining the platform model without changing the application model. It then becomes possible to annotate the cost of the execution with the given task in the application model and, so, it adds a quantitative cost measure for use in the assessment of the platform.

It is a requirement that a given mapping is valid in order for the resulting system model to be used for performance estimation through simulations. A mapping is said to be valid, if and only if, all the requested services of a given application model are offered by the processing elements of the platform model onto which it is mapped. However, it is *not* a requirement that all tasks of an application model are mapped to the processing elements of a platform model. In the case where a task of an application model is unmapped, only the functional

behaviour of the tasks is modelled and no performance estimates are associated with that particular task. Currently, the validity of a mapping is determined during runtime only, resulting in an error during simulation in the case of an invalid mapping. In the future, however, tools for performing such checks prior to runtime will be constructed. Also, future work will include the possibility of performing checks for the semantic validity of specific mappings to ensure, e.g., that no dead locks occur in a given mapping.

If multiple service models of the application model are mapped to the same service model of the platform model, a scheduler should be provided as part of the platform model. Schedulers are implemented as separate service models e.g. acting as abstract operating systems. When the tasks of an application model are mapped to the processing elements of an application model this is done by mapping the active interfaces of the tasks of the application model onto the passive interfaces of the processing elements of the platform model.

The use of services implies that a high level of flexibility is obtainable and that it is possible to perform high level performance estimates using services with a high level of granularity only, ranging to low level execution of the actual implemented applications. The required services of the application model under consideration can be used to guide the selection of platform components from a library. In this way the required services of the application model can be used to synthesize the target architecture.

4.4 Summary

In this chapter one of the main contributions of the work that has been carried out during the course of this Ph.D. study - the system level modelling and performance estimation framework - was presented.

The framework is related to what is known as the Y-chart approach [12, 50] and leverages principles of the Platform Based Design (PBD) principle [48]. The functional requirement of applications is captured by application models specifying the functional behaviour and abstract inter-task communication requirements. The target architecture is modelled by a platform model possibly composed of both hardware and software components. The application model is then related, through an explicit mapping step, to the platform model forming a system model. Quantitative performance estimates can then be produced through simulation.

Application models are used to capture the functional behaviour of the application considered. Preferably described in such a way that inherent task-level parallelism is exposed making inter-task communication requirements explicit. It is very much desirable to have application models describe without any as-

sumptions of the implementation in order to maximize the possible platforms for execution of the application.

Platform models, on the other hand, are used to capture the implementation details, including the cost, of a particular target architecture. The target architecture can be used to associate a cost with the execution of a given application on the specific architecture. However, it is also possible to model the actual implementation specific functionality in the platform model, in this way reflecting the actual implementation, without having to change the application model. And, so, the same application model can be modelled as executing on many different platform models very easily, allowing these to be compared based on a desired cost metric chosen by the designer using the framework.

Quantitative performance estimates are produced through the simulation of a system model. It is the system model which, through an explicit mapping of the components of an application model onto the components of a platform model, brings the two models together and allows a modelling of the execution of the application on a specific target architecture.

Fundamental to the framework is the service model concept which allows a unified modelling of embedded systems and which supports a gradual refinement of abstraction levels and allowing components described at different abstraction levels to co-exists. Service models are used for modelling components of both the application and target architecture. Through the use of services it is possible to achieve a decoupling of the specification of functionality, communication, cost and implementation. The notion of services plays a vital role in the system level performance estimation framework presented. Services are used to represent functionality of the components of which a system is composed. It is the notion of services which is the enabling factor for the unified approach presented here for modelling complete systems composed of both hardware and software elements. Services offered by a service model may themselves be requiring the presence of other services, i.e. offered by other service models. The use of services to abstract away implementation details implies that there is no distinction between components actually being implemented in hardware or software. In the early design phases this is of great benefit allowing the design space to be explored without the constraints of an unjustified co-design choice of whether the component is implemented in hardware or software. The co-design problem can be addressed based on actual quantitative performance estimates such that the decision of implementing a component in either hardware or software can be based on a sound and well-founded basis.

Part II

Realization and Usage

Chapter 5

Realization of the Framework for System Level Modelling and Performance Estimation

In this chapter an overview is given of the currently realized framework developed during the course of this Ph.D. study. The framework implements the concepts presented in the previous chapters and includes a custom simulation engine and a number of tools and libraries for supporting the practical use of the framework. The focus of the chapter is on the practical aspects of using the framework.

The objective of having a realization of the framework is to allow a proof-of-concept based on a practical evaluation. Thus, it was of the utmost importance to have maximum flexibility of the underlying simulation engine as well as the possibility of modifying and controlling all aspects of this. Therefore, it was decided to implement a custom simulation engine instead of using e.g. SystemC and the discrete event simulation engine supplied with the SystemC libraries.

However, during the last couple of years, as SystemC have gained even more widespread adoption throughout the industry, especially after the TLM 2.0 standard was released, SystemC is becoming more and more attractive for the implementation of a framework such as the one presented in this thesis. The restrictions of a proprietary framework as the one presented in this thesis severely limits a widespread use. The presented framework, however, is by no means

bound to the current implementation and an investigation of an implementation of the current framework in e.g. SystemC could (should) be performed in the future.

5.1 The Current Implementation

The current implementation of the framework is done in Java [73] and models are specified directly in Java as well through the use of a number of libraries developed for supporting the concept of service models, application models, platform models and system models. In addition to the libraries a simple graphical user interface has also been implemented.

```
<?xml version="1.0"?>
<system>
  <application name="ApplicationX" id="application"
    class="..ApplicationModelX">
    <serviceModel id="application.task0" class="...Task0" />
    <serviceModel id="application.task1" class="...Task1" />
    ...
    <serviceModel id="application.taskN" class="...TaskN" />
  </application>

  <platform name="PlatformX" id="platform"
    class="..PlatformX">

    <serviceModel id="platform.pe0" class="...ASIP">
      <argument id="FCLK" value="25" />
    </serviceModel>

    <serviceModel id="platform.pe1" class="...DSP">
      <argument id="FCLK" value="50" />
    </serviceModel>
  </platform>

  <mappings>
    <mapping sourceId="application.task0" targetId="platform.pe0">
      <interface active="EXECUTE" passive="TASK" />
    </mapping>

    <mapping sourceId="application.task1" targetId="platform.pe1">
      <interface active="EXECUTE" passive="TASK" />
    </mapping>

    <mapping sourceId="application.taskN" targetId="platform.pe2">
      <interface active="EXECUTE" passive="TASK" />
    </mapping>
  </mappings>
</system>
```

Listing 5.1: System model configuration in XML.

The graphical user interface is built on the Eclipse [32] platform. It allows simulations to be controlled and inspected in a more convenient way than pure text based simulations.

Java as implementation language brings several benefits with respect to portability of the implemented simulation framework; however, Java is not optimal for specifying models as done currently. The development of a custom language as specification language to address this issue has been considered but it has not been deemed of vital importance to the current objective of the framework in which the focus is still mainly on a proof-of-concept.

The core implementation provides classes for describing service models, application models and platform models as well as a simulation kernel which allows system models to be simulated. In addition to the core components, a number of utility interfaces and classes are used to provide unified access to elements of the models in order to allow a graphical user interface to present information about e.g. a service model.

The composition of a system model is specified using a XML configuration file. The configuration file specifies the application model and platform model to use, the components of each which are to be used as well as the mapping of application model components onto platform model components. The use of a configuration file provides an easy way for configuring system models through simple declarative constructs. An example of a configuration file is shown in listing 5.1.

Figure 5.1 shows a screen shot in which a system model is being simulated. The system model of the example is composed of four service models which are named **i2c, source, sink** and **svf0** as can be seen in the **debug**-view.

In this case the model is used as an instruction set simulator supporting breakpoints, single stepping through service requests and general simulation. The service request currently being requested to be executed in the processor service models includes a reference to the corresponding assembly level source code allowing this to be displayed in the graphical user interface as well. The framework also provides a view of any declared state holding elements of a service model through the graphical user interface. In this way the registers and memory elements of the processor model may be displayed and inspected in the graphical user interface.

In order to support a debugable version of the core, special debugable versions of all components of the models have been constructed. These can be seen as wrappers of their corresponding components and, through a number of interface defined member methods, provide access to information about the given model. The debugable wrappers are transparent to the designer and, thus, the designer needs only implement a model once. The separation of models into debugable and normal versions allow designers to start regular simulations in which no per-

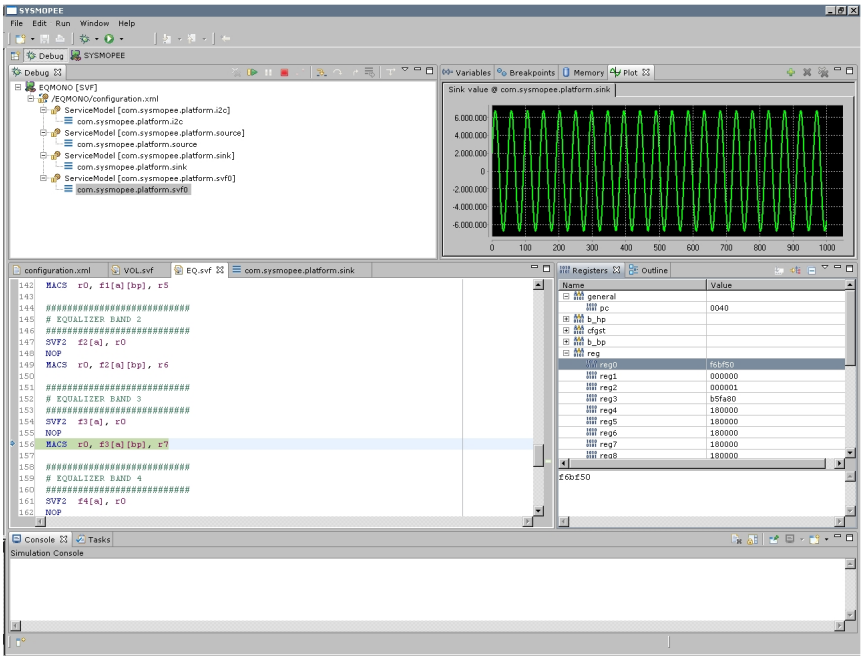


Figure 5.1: Screenshot of the graphical user interface.

formance degradation is experienced during simulation or to start a debugable simulation in which one can single step through the simulation and inspect the individual instantiated components of the system model under simulation which, in this case, would result in a performance degradation.

5.2 Simulation Engine

This section describes the simulation engine used for simulation of the models constructed in the presented system level performance estimation framework in order to obtain quantitative performance estimates.

In order to support the existences of multiple different models-of-computation, a discrete event simulation engine has been chosen for coordinating the execution of the individual service models and their internal processes. Discrete event modelling is used within a range of different application domains and described thoroughly in e.g. [17].

A custom simulation engine for proto-type use supporting the modelling framework presented has been implemented during the course of this project. The

reason for implementing a custom simulation was to obtain maximum control over the simulation kernel in order to be able to validate the ideas of the framework. The implemented discrete event simulation kernel keeps track of simulation time and schedules the execution of the individual processes of the service models according to their sensitivity to events.

As already described in section 3.4, a number of events are associated with the lifetime of a service request which by their occurrence can trigger processes sensitive to the event. The occurrence of a given event allows a waiting process to become activated at different phases of the simulation as described by the designer of the model and depending on the desired behaviour.

The simulation engine provides basic semantics for controlling the execution of a service model through a process like behaviour for modelling concurrency and a number of waitFor-statements as listed in table 5.1 which will cause the execution of the model to block.

waitFor(time)
waitFor(service request, event type)
waitFor(interface, service request type, event type)
waitFor(event)

Table 5.1: Process wait-types.

waitFor(time) causes the process to block until the specified time has passed. The effect is that an event is being scheduled with a time tag of the current simulation time plus the specified time value. The simulation engine will fire the event when the specified simulation time has been reached and the process will resume its execution.

waitFor(service request, event type) causes the process to block until the specified event type of the specific service request instance is being fired. When this occurs the process will resume its execution and the time tag of the event will be decided by the simulation engine.

waitFor(interface, service request type, event type) causes the process to block until the specified service request type and event type is being fired at the specified service model interface. In this case the simulation time will also be annotated by the simulation engine when the event is fired.

waitFor(event) causes the process to block until the specified event is fired. When the event is fired the process will be notified and continue its execution.

5.2.1 Representation of time

The simulation engine is implemented as a discrete event simulation engine using a delta-delay based representation of time. The delta-delay based representation divides time into a two-level structure: Regular time and delta-time. Between every regular time interval, there is a potentially infinite number of delta-time points $t + \delta, t + 2\delta, ...$ Each event is marked with a time tag which holds a simulation time value which indicates when the event is to occur. Every time an event is fired and new events are generated having the same regular time value as the current time value of the simulation, e.g. in the case of a feedback loop, the new event will have a time stamp with the same regular time value but now with a delta value incremented by one. The use of delta-delays ensures that no computations can take place in zero-time, but will always experience minimum a delta-delay. The delta-delay based representation is making simulations deterministic because the use of delta-time makes it possible to distinguish between which of two events generated at the same point of time is to be processed first by looking at their delta-value.

Application models have no notion of time. It is only when the application model is mapped to a platform model that it becomes possible to annotate the execution time of the application model by relating the execution of its tasks and the generation of service requests to discrete time instances.

In the platform model, on the other hand, time is represented explicitly using the delta-based representation of time. Each model of the platform can be modelled with arbitrary delays or specify a clock frequency at which they want to be evaluated. In this way, it is possible to model synchronous components in the platform model which are only activated at regular discrete time instances. Thus, the tasks of the application model are blocking while the service requests are being processed in the platform model and, so, makes it possible to associate an execution time metric with tasks. Similarly, it is possible to annotate other types of metrics such as power costs, etc.

A special event *type* is used to represent hardware clocks used e.g. by models described using clocked synchronous models-of-computation. These models do not use time explicitly, instead represent time as a cycle count, however, in order to be used in the currently implemented simulation engine, they still need to specify a clock frequency which determines how often they are allowed to evaluate. Regular events are removed from the event list, executed and then disposed and because events are bound to a unique time instance they can thus only occur once. However, the special event type used to represent hardware clocks is implemented as a re-schedulable event in the simulation engine which takes care of handling the uniqueness of each instance of this event type. Such a clock event is automatically rescheduled and re-inserted into the event list. Each clock event object has a list of active processes which are to be evaluated

when the clock ticks. This list is updated dynamically during simulation, and in the case where a clock has no active processes in the active list, the clock itself is removed from the pending list of events in order to increase simulation performance. When a clock event object is inserted into the event list, it will be inserted sorted according to the simulation time of the next clock tick of the clock event object. It is also possible to have a clock object which contains no static period. In this case, the clock ticks can be specified as random time points or as a list of periods which can be used once or repeated.

Of course, the platform model can also contain service models which are activated on the arrival of service requests only triggered by the event associated with the request of the service. In these cases, the occurrence of such an event, at a specific simulation time, will activate the blocking process, also allowing a modelling of e.g. a combinatorial delay if needed. The modelling of combinatorial elements can also be handled in zero regular time, in this case experiencing only a delta-delay.

5.2.2 Simulation

The simulation engine uses two event list for controlling the simulation: One for delta events and one for regular events. The delta event list contains only events with a time tag equal to the current simulation time plus one delta cycle. The regular event list on the other hand contains pending events with a time tag in which the regular time is greater than the current simulation time. The regular event list is sorted according to the time stamp of the events in increasing order. In this way the head of the regular event list always points to the event with the lowest time tag.

The simulation engine always checks the delta event list first; if the list is not empty, the delta cycle count is incremented and all events contained in the delta event list are fired one by one until the list is empty, implying that all events belonging to the same delta cycle have been fired.

If no delta events are pending, i.e. the delta event list is empty, the simulation engine detaches the first event in the regular event list and advances the simulation time to the time specified by the time tag of the event. Also, the current delta count is cleared and the event is then fired.

Delta-delay based discrete simulation engines suffer the risk of getting stuck in infinite loops where time is not advanced and only the delta cycle is incremented. A naive approach to handle this is implemented, allowing the designer to specify a maximum number of delta cycles to be allowed before the simulation engine quits the simulation.

The firing of an event can of course cause new events to be generated and

scheduled in the simulation engine. If a new event is generated having its time tag set to the same regular time value as the current simulation time, it is added to the delta event list; otherwise it is scheduled according to its time tag and inserted into the regular event list according to the time tag specified. The delta-event list only contains events with a time tag equal to the current simulation time plus one delta cycle.

After an event has been fired it is checked if the event type of the event is periodic. Events belonging to a periodic event type, e.g. the special clock event object described in section 5.2.1, are then automatically rescheduled and inserted into the event list at the correct position.

5.3 Producer-Consumer Example

In order to illustrate the usage of the framework and elaborate more on the different elements, a simple producer/consumer application is considered in the following.

The first step in order to start using the presented framework is to construct an application model. The application model captures the functional behaviour of the application in a number of tasks as well as specifies the communication requirements of the individual tasks explicitly, without any assumptions on the implementation, following the principle, on which the framework is founded, of separating the specification of functionality, communication, cost and implementation. The application model serves as the functional reference in the refinement steps towards the final implementation. However, at this level of abstraction, there is no notion of time or physical resources - hence only very rough performance estimates can be generated from a profiling of the application model.

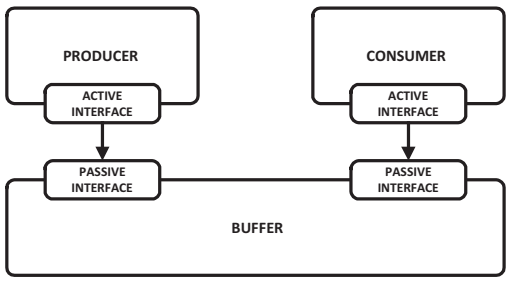


Figure 5.2: A simple application model consisting of a producer and a consumer communicating through an abstract buffer.

Figure 5.2 shows an application model composed of a consumer task and a pro-

ducer task which communicate through an abstract buffer. Both the tasks and the abstract buffer are modelled by service models. The producer has an active interface which is connected to the passive write interface of the buffer which offers a **write** service. Similarly, the consumer has an active interface which is connected to the passive read interface of the buffer model which offers a **read** service. The producer and consumer models are the only active models, i.e. only the producer and consumer models can initiate the request of services.

The service models are specified as one or more concurrently executing processes. Processes execute until they are blocked, waiting for some condition to become true. When the condition becomes true, the process can continue. Such behaviour can be implemented using threads. The thread-context switching required, each time a process is being blocked or activated, has a high impact on simulation performance and due to the fact that we currently use a global notion of time in the discrete event simulation engine, most processes will execute in a lock-step fashion, blocking and unblocking requiring context switches very often. At the same time, threads provide more functionality than actually needed now that only one process can be active at a time in the *physical* simulation engine. Thus, the desired behaviour is found in the much simpler concept of co-routines which allows execution to be stopped and continued from the point where it left off. The current implementation of the simulation kernel thus used a concept very similar to co-routines implemented in Java [73]. This requires that the designer of a process must divide the code body of the process into blocks. Blocks are executing sequentially and the execution of a block cannot be stopped. When the code contained in a block is done executing, the block returns a reference to the next block to be executed. In this way it is possible to model the blocking of a process on some condition and then when the condition becomes true, the process will carry on its execution by executing the block of code returned by the previous executing block. The full source code of the functional application model of the producer-consumer example can be found in appendix A.

Listing 5.2 shows the description of the main body of the producer service model. In this case a single process is used to capture the behaviour of the producer. A similar description is made for the consumer which, however, is not shown. The main body of the producer is actually an infinite loop in which the producer first calculates some value, then instantiates a **write** service request with the calculated value as argument and then requests the **write** service via an active service model interface. It is possible to request a service and then continue the execution directly, however. In such a case a waitFor-statement is used to block the producer service model until the requested write service has been executed, signalled by the firing of a service done event. When the service done event is fired, the producer will be activated and rescheduled for execution in the simulation engine. The producer will then resume execution continuing from the point at which it was blocked.

```
...
private IActiveServiceModelInterface fWriteInterface;
...
public final IBlock execute(){
  // Calculate data
  ...

  // Create write service request
  IServiceRequest sr = fWriteInterface.createServiceRequest(
    "WRITE", new Object[]{...});

  // Request write i.e. production of data
  fWriteInterface.request(sr);

  // Wait for write request to be done
  waitFor(sr, IServiceRequest.EventType.DONE);

  ...
}
```

Listing 5.2: Body of the producer service model main process.

As can be seen, the **write** service request is both used for signalling the request of a **write** service but also to transport the actual data values which are to be exchanged between the producer and buffer service model and which will be written into the buffer, eventually. In this way, arbitrary data and objects can be transferred.

The buffer model is only activated when a service request is requested through one of its passive interfaces. In such a case, the behaviour of the **read** and **write** services depends on the implementation of the buffer and, so, e.g. different blocking or non-blocking read and write schemes can be investigated easily simply by interchanging the buffer model.

As an example, the buffer service model described in listing 5.3 uses a blocking write policy. Again, the main body is actually executing an infinite loop. The main body starts executing a waitFor-statement, blocking the execution until the arrival of a **write** service request, indicated by the firing of a service request requested event of type **write**. In this case the waitFor-statement is not instance sensitive as in the case of the producer which was blocking for a service request done event on the instance of the requested **write** service request. Instead it is blocking until a service request requested event of the specified type is fired indicating a request through the active service model interface. This implies that whenever a service request requested event of type **write** is fired, the write process of the buffer service model will be scheduled for execution in the simulation engine and the write process will become active and continue its execution from the point it left off. It then starts the actual execution of the requested **write** service. If the buffer is already full, the write process will block

once more until an empty slot in the buffer becomes available. Otherwise, the actual write to the buffer will be executed and a service request done event will be fired in order to notify the requester, in this case the producer, that the service request has been executed. Similarly, a read process implementing a blocking read policy is described in the same manner; this, however, is not shown here.

```
...
private IPassiveServiceModelInterface fWriteInterface;

private IServiceRequest fWriteServiceRequest;

public final IBlock execute(){
  // Wait until a write service is requested
  waitFor(fWriteInterface, "WRITE",
    IServiceRequest.EventType.REQUESTED);

  ...

  if(fFull){
    // Wait until notified
    waitFor();
    ....
  }
  else{
    // Perform write
    ....

    // Signal done
    fWriteServiceRequest.done();
    ....
  }
}
```

Listing 5.3: Main body of the write process of the buffer service model.

In order to generate quantitative performance estimates, the task and buffer service models of the application model must be mapped to the service models of a platform model creating a system model. Performance estimates relevant for evaluating the different platform options can then be extracted from the simulation of the system model. When the service models of an application model are mapped to the service models of a platform model, the service models of the application model can, when executed, request the services offered by the platform service models. In this way, the functionality of e.g. a task is represented by an arbitrary number of requests to services which, when executed, model the execution of a particular operation or set of operations. The execution of a service in the platform model can include the modelling of required resource accesses and latency only, or, depending on the level of abstraction used to describe the service model, even include the actual functionality including bit true operations.

The mapping of the service models of the application model to the service models of the platform model need not to be complete, i.e. it is allowed that only a subset of the service models of the application model are mapped. In this case, the unmapped service models of the application model captures all functionality, i.e. both the control flow and data operations within the application model, and have no costs associated. In figure 5.3, the consumer service model is an example of an unmapped service model.

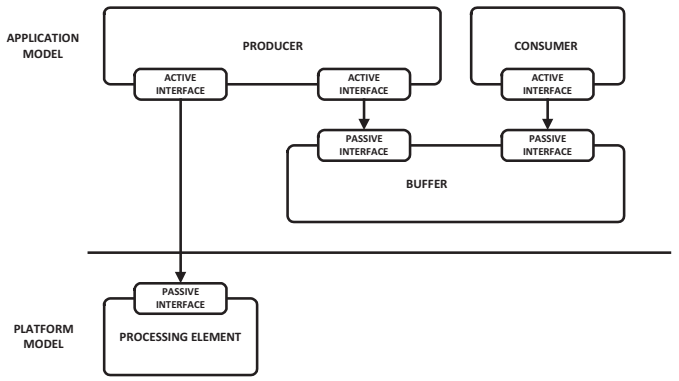


Figure 5.3: Only the producer task is mapped to the processing element of the platform model. The consumer task is unmapped and modelled functionally only.

On the other hand, the service model of the producer in figure 5.3 is an example of a mapped task. As a first refinement step, the service model of the platform model, to which the producer service model is mapped, is only used for latency modelling and for associating a cost with the execution of the producer. This is achieved by modelling a processing element which offers the service **produce** through a passive service model interface.

```
IActiveServiceModelInterface fExecute;
...
// Calculate data
IServiceRequest sr = fExecute.createServiceRequest(
  "PRODUCE", new Object[]{...});
fExecute.request(sr);
..
```

Listing 5.4: The refined production of data values, now depending on the platform model mapping.

The producer service model, as shown in listing 5.4, then requests the **produce** service via its active service model interface during execution of the **produce** service, modelling a particular implementation in terms of cost and latency. In

this example, the **produce** service is a very abstract service; however, it is the designer of a model who determines the level of abstraction used to associate with each service. One could also imagine the specification of required services for each arithmetic operation required for calculating the produce value. Currently, required services are specified manually and in order to have a valid mapping of a service model from the application model to the service model in a platform model, all required services must be provided by the service model of the platform model. The higher level of abstraction used to specify required services also implies that there are more options for mapping a model because the separation of functionality and implementation is retained.

If preferred, it might also be possible to refine the processing element to include the calculation of actual data values as well, in this case using a mixed task, modelling part of the behaviour of the producer according to the functional specification in the application model and part of the behaviour according to a particular implementation in the platform model and, so, mix different levels of abstraction seamlessly. This is particularly useful in the early stages of the design process where rough models of the platform might be constructed. In such a scenario, fast estimations of the effect of adding redundant hardware support for specific operations, or even rough estimates of the effect of using multi-processor systems, the effect of buffer sizes etc. can be explored. This scenario might be refined to a level where cycle accurate and bit true models are described, but still leaving the control flow of the application to be handled in the application model and modelling only the cost of control operations in the platform model.

Such an example is seen in figure 5.4, where the producer-task could be implementing e.g. loop control in the application model directly, and only part of the functionality of the loop body through requests to services offered by the service model of the processing element in the platform model onto which it is mapped. Figure 5.4 also shows how it is possible for the producer, mapped to and executing on the processing element, to communicate through a partially mapped buffer with the functional consumer service model which is only modelled in the application model.

Finally, as shown in figure 5.5, it is possible to model a full mapping of an application model to a platform model including both the active service models, in this case the producer and consumer running on one or more processing elements, and the passive service models, in this case the buffer mapped to a block of memory. Still, it is possible to mix partially functional behaviour modelled in the application model with the real behaviour of the implementation modelled in the platform model. In the most extreme case, the producer and consumer is represented solely by a set of requests to the services offered by the processing element of the platform model onto which it is mapped. In this case the complete functionality is modelled in the platform model, implying that the

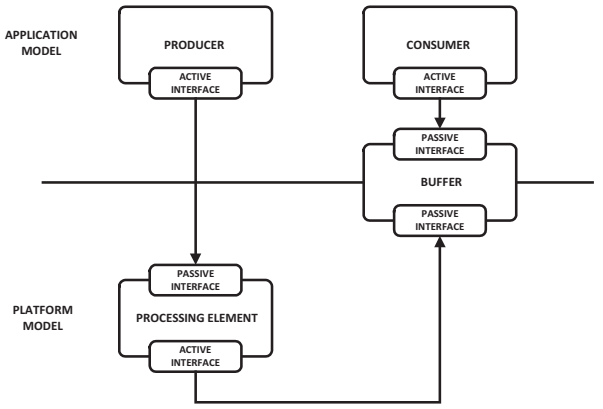


Figure 5.4: Only the producer task is mapped to a processing element of the platform model. The consumer task is unmapped and modelled functionally only. Similarly, the FIFO buffer is only partially mapped, allowing it to be accessed from the active interface of the processing element in the platform model and at the same time being accessible from the active interface of the consumer task in the application model.

producer and consumer are modelled as tasks belonging to the implementation category. Consequently, the complete task is represented by a service request image, directly equivalent to a binary application image of e.g. a processor. Another advantage of the support for such compiled tasks is that a platform model described at this level of abstraction can also be used for performance estimation of compiler technologies.

5.4 Summary

In this chapter an overview of the implementation of the presented system level modelling and performance estimation framework in Java was presented.

Currently, an implementation of the framework exists at a proof-of-concept level consisting of a core implementation of the framework and a graphical user interface. The framework can be used in console mode in which simulations are performed without any interaction with the user or a graphical user interface built on the Eclipse platform can be used. System models being simulated are configured through an XML-configuration which dictates the components to be used in the system model and their configuration. The graphical user interface allows an easier debugging of models by providing basic debugging capabilities such as single stepping through a simulation. Also, the graphical user interface

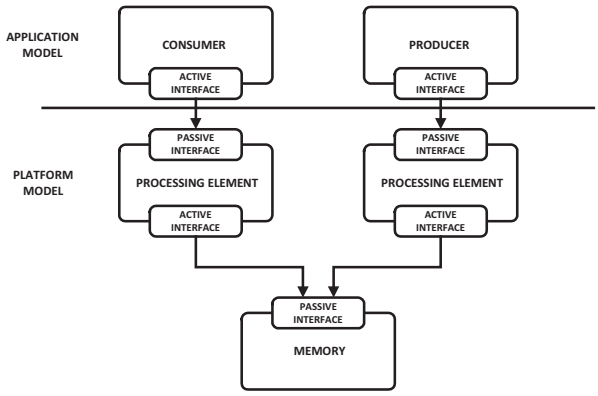


Figure 5.5: Both the producer and consumer tasks are now mapped to processing elements of the platform model. The FIFO buffer is mapped to the service model, modelling a block of shared memory to which both processing elements have access.

makes it possible to inspect the variables and state of a model.

As the complete framework is implemented in Java, extensions can be made within the scope allowed by the Java language. This also implies that any Java debugger can be used to inspect the low level details of a simulation. However, when a more complete version of the support tools for the framework is implemented in the future, the implementation language should be transparent to the user of the framework and thus Java need not to be the chosen language for implementation.

The objective of the realization of the framework in Java was a proof-of-concept, allowing the concepts introduced in chapter 3 and 4 in order to allow the presented ideas to be assessed and evaluated in practice. The proprietary implementation of the framework in Java was chosen over an implementation in e.g. SystemC based on a consideration of an academic nature, namely, in order to have full control of all aspects of the framework. The proprietary implementation in Java, of course, has the limitation of requiring all models to be defined within the syntactical requirements imposed. However, Java is merely an implementation language, hence other languages or possibly even other simulation frameworks could have been used for realizing the framework in practice. Especially, SystemC and in particularly the transaction level modelling paradigm defined in TLM 2.0 would be an interesting alternative implementation which might prove more practical in order to obtain a wide spread use of the framework.

Chapter 6

A Service Based Model-of-Computation for Architecture Modelling and Simulation

In order to allow efficient system level design of embedded systems, a flexible modelling of hardware components allowing fast and accurate performance estimation to be carried out is required. Several methods have been presented in recent years allowing performance estimation through formal analysis or simulations of architectures at high levels of abstraction [14, 20, 36, 56, 68, 87].

Recently, approaches that rely, at least partly, on formal methods of analysis in order to allow performance estimation have been presented. In theory, these approaches eliminate the need for simulations in order to predict performance. However, in most cases, the accuracy of these approaches only justifies their use in the very early stages of the system design phase where they can be used to reduce the number of potential candidate architectures, as is done in [56], before more time consuming detailed performance estimates, obtainable only through simulation, are produced in the later design stages.

The majority of the approaches based on fast simulations, e.g. [3, 36], are using high speed functional instruction set simulators with high level modelling of data memories, caches, inter-connect structures, etc. performing a number of

abstractions thereby trading accuracy for simulation speed. These approaches have their merit, especially in the early design stages, and often even allow software developers to begin the target specific software development in parallel with the hardware developers - long before low level register transfer level descriptions of the platform exist or the actual hardware prototype bring-up.

The high level models fulfil the needs for early software development and initial architectural exploration. However, in many cases one needs to be able to generate accurate performance estimates in order to reason about the actual performance of the system so as to verify architectural design choices. In order to do so, cycle accurate models are required, implying that, currently, register transfer level descriptions of the architectural elements of the target platform are often the only viable solution. The simulation of large scale systems described at the register transfer level, however, suffers from tremendous slowdown in the simulation speed compared to the high level simulations. Even worse, the development of such detailed descriptions is long and costly implying that when these are finally available, often at a very late stage of the development phase, changes of the architecture are very hard to incorporate resulting in limited possibilities for design space exploration.

Thus there exists a gap between the fast semi-accurate methods which are highly useful in modern design flows allowing the construction of high level virtual platforms, in which rough estimates of the performance of the system can be generated, and the detailed and very accurate estimates which can be produced through register transfer level simulations.

In this chapter a model-of-computation fitting into the service-based system level modelling and performance estimation framework will be presented which allows a flexible modelling of synchronous components at different levels of abstraction covering initial high level models to cycle accurate and bit true descriptions.

The model-of-computation presented in this chapter is based on Hierarchical Coloured Petri Nets (HCPN) [44] and is targeted at the modelling of synchronous hardware components only. The HCPN based model-of-computation presented here is taking its point of departure in the initial work presented in [38] which was only intended for modelling a single top level hardware component and, thus, could not be used in a compositional modelling framework as the one presented in part I. This issue is addressed in the method presented in this paper and the basic approach is extended heavily both in terms of applicability and in terms of simulation efficiency, making the HCPN based service models usable in the current context.

In the following, the basic concepts of the HCPN based model-of-computation will be presented, then a number of optimizations which can be performed due to the restricted modelling domain of synchronous hardware are introduced

which allow efficient simulations as will be shown in section 6.3. It should be noted that, as was the case of the framework for system level modelling and performance estimation introduced in part I and II, the current implementation of HCPN based service models is done in Java [73].

6.1 The HCPN based model-of-computation

The HCPN based model-of-computation for modelling synchronous hardware components is an example of a clocked synchronous model-of-computation which can be used in the presented system level modelling and performance estimation framework.

HCPN has been selected due to the great modelling capabilities with respect to concurrency and resource access as well as the compositional properties which match the requirements of service models very well. Also, the formal analysis methods which are very mature for HCPN, regarding e.g. deadlock and reachability analysis makes HCPN very interesting. The formal analysis capabilities, however, are not investigated in this thesis and thus belong to the category of future work.

Traditionally, HCPNs have been used mostly for high abstraction level modelling and only been in limited use for describing cycle accurate hardware models. This is most likely caused by the complexity of managing the synchronous execution of transitions using the general HCPN model-of-computation.

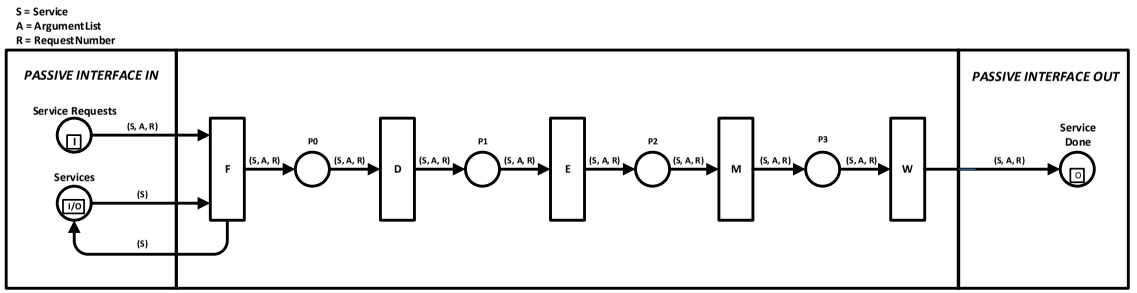
In order to introduce the basic concepts of the modified HCPN based model-of-computation for describing synchronous hardware components, the ARM9TDMI processor core [66] is considered in the following. The ARM9TDMI processor core is a 32-bit RISC like processor which implements the ARMv4T [67] instruction set. It has a 5-stage pipeline and 16 general purpose registers visible at a time, connected to a shifter, arithmetic logic unit, and a multiplier. The ARM9TDMI processor is a Harvard architecture, hence access to data memory and program memory can be done in parallel.

Service models described using the modified HCPN model-of-computation is constructed in a way very similar to traditional HCPNs. Through the use of places and transitions connected by arcs, a model of the target component is composed. Arcs have associated arc expressions which determine the tokens produced and consumed when the arc expressions are evaluated during the firing of a transition. Transitions have associated guard expressions which are Boolean expressions that must evaluate to true in order for the transition to become enabled. In addition to transition guards, our modified HCPN model also supports place guards. Place guards are equivalent to transition guards and, thus, must evaluate to true before a transition is enabled which is connected

to the place through an outgoing arc with an arc expression that requires the production of tokens into that place. Transitions become enabled and can fire when the tokens specified by all ingoing arc expressions are available, i.e. all variables of the arc expressions can be bound, the transition guard expression evaluates to true, and when the destination place guards of the outgoing arcs of the transition also evaluate to true. Such a binding of variables is referred to as a binding element. Transitions can also have an action associated which specifies a piece of functionality which is carried out when the transition is allowed to fire. Structural hierarchy is supported directly through the use of substitution transitions as known from regular HCPNs. The use of substitution transitions also allow a gradual refinement of models in which different sub-blocks of a model can be easily exchanged.

In contrast to generic HCPN, only two types of tokens exist in the modified HCPN model-of-computation. The first representing services which are used to model resources and the second representing service requests which are used to model access to resources. The HCPN based service model also defines three special places which are used to support the interface-based approach used by the service model framework. These are: The **Service Request** place, the **Service Done** place and the **Service** place. Inter-model communication is handled through the exchange of service requests via active-passive service model interfaces following the semantics introduced in section 3.4. The initiator model can request one of the services specified, as required by the model, without knowing any details about the model which provides the required service or how it is implemented. When a service request is made to a HCPN based service model, a service request token is produced in the service request place associated with the interface. If the specified service is available in the **Service** place, the service request will start its execution. When the specified service request has been executed, the service request will arrive in the **Service Done** place, associated with the passive interface from which it was requested, signalling the completion of the service request. When this happens, the initiator model (i.e. the model which requested the service) is notified that the execution of the requested service is done.

Returning to the ARM9TDMI processor core, a graphical representation of the HCPN model of the processor is shown in figure 6.1. In the HCPN based service model, the 5-stage pipeline is clearly identifiable. The 5-stage pipeline consists of the instruction fetch stage (F), the instruction decode stage (D), the execute stage (E), the data memory access stage (M) and the register write stage (W). In the figure, the associated transition actions for an implementation of the data processing instructions of the ARM9TDMI core are illustrated using pseudo-code. The actions allow arbitrary functionality to be associated with a pipeline stage and thus the amount of details included determines the abstraction level used allowing a modelling of high level functionality only to cycle accurate and actual bit true modelling.




```
transition F{
  bool guard(){
    return p0.isEmpty();
  }

  void action(){
    _PC += 4;
  }
}
```

```
transition E{
  bool guard(){
    ...
    if(_DINST.contains(_SR.S)){
      return _WT.contains(_SR.RN)
|| _WT.contains(_SR_SOP);
    }
    ...
  }

  void action(){
    ...
    if(_DINST.contains(_SR.S)){
      ((DINST)_SR).execute();
    }
    ...
  }
}
```

```
transition W{
  bool guard(){
    return true;
  }

  void action(){
    ...
    if(_DINST.contains(_SR.S)){
      _REG[_SR.FRD] = _ALU_OUT;
    }
    ...
  }
}
```

```
transition D{
  bool guard(){
    return p1.isEmpty();
  }

  void action(){
    ...
    if(_DATA.contains(_SR.S)){
      _SR.RN = _REG[_SR.FRN];
      _SR.SOP = getShifterOp();
    }
    ...
  }
}
```

```
transition M{
  bool guard(){
    return p3.isEmpty();
  }

  void action(){
    ...
  }
}
```

Figure 6.1: HCPN model of ARM9TDMI processor core which is an implementation of the ARMv4T instruction set.

The instructions of the ARM9TDMI processor is represented by a number of services and initially, one token for each service provided is put in the place **Services**, e.g. one token representing addition **ADD**, another token representing multiplication **MUL** and so on.

The primary purpose of the HCPN model is to capture the parallel execution of hardware. This implies that state holding elements, of e.g. a processor, are not represented directly in the HCPN model but are simply contained in variables of the model. These state holding elements, however, are only changed when a transition is allowed to fire modelling synchronous behaviour. This is also the case in the ARM9TDMI processor example in which registers are modelled as variables in the model which, e.g. in the case of data processing instructions, are read in the decode stage (transition **D** in the figure) and written in the write back stage (transition **W** in the figure). It is also possible to model pipeline interlocks; in the example this is done in a combination of using tables holding information about reserved registers (i.e. registers which are to be written, and in which no forwarding of data operands is possible) and guard expressions. When a transition guard expression evaluates to false, the transition is not allowed to fire, corresponding to an insertion of bubbles into the pipeline from the instruction decode stage (transition **D** in the figure).

To summarize, the basic structure of the HCPN based service model implementation dictates the latency, resource requirements and concurrency properties of each service offered, whereas the actual behaviour of the services is implemented by associating actions with the transitions of the model. If transition actions are combined with the possibility of adding arguments to the service requests, the actual behaviour or functionality of the service can be implemented allowing e.g. the implementation of an actual addition of two values or the possibility of modelling data operand dependencies, etc. This emphasizes the great potential of the method with respect to flexibility and accuracy. It is indeed possible to refine models to a level where they can be used for e.g. cycle accurate instruction set simulators if needed. Also, it is possible to associate a quantitative cost measure which each service. The cost measure can be pre-computed or computed dynamically during runtime and, so, provides the user with a quantitative cost measure for the execution of a given application on the model described.

6.2 Simulation Model

In this section an introduction to a number of optimizations which can be performed to the models described using the modified HCPN model-of-computation prior to runtime will be given in order to achieve efficient simulations. These optimizations significantly boost the simulation performance allowing performance estimates to be generated efficiently.

The HCPN based model-of-computation presented here shares properties with the Synchronous Data Flow model-of-computation [60] because the modelling domain is constrained, modelling synchronous hardware components only. However, an important difference is that the HCPN based model-of-computation supports choices and conditional evaluations in the search for enabled transitions as well as in the consumption and production of tokens and hence the transition firing sequence cannot be scheduled statically. However, the modified HCPN model-of-computation allows a quasi-static scheduling of the firing sequence of the transitions of the models to be used. The idea of quasi-static scheduling [23, 95] in this context is to perform a static analysis of the model in order to make as many choices as possible prior to runtime, leaving only the data dependent dynamic choices to be evaluated at runtime.

Synchronous hardware components are modelled by firing all concurrently enabled binding elements of a model in each simulation cycle, implying that all transitions, if enabled, must fire exactly one time in each simulation cycle. Combined with the fact that the model structure does not change during runtime, it is therefore possible to extract a static schedule which determines the order in which the transitions are allowed to fire prior to runtime. At runtime, transitions are evaluated for enablement according to the extracted schedule, forming a quasi-static scheduling of the firing of the transitions of the model. The schedule is determined by sorting all transitions of a service model implementation with respect to the longest topological distance to the service done place of the model. If two transitions have the same topological distance to the service done place, and they are not independently enabled, i.e. they share an input place, a priority must be assigned by the designer of the model to resolve such ambiguities. In the case of the service model of the ARM9TDMI processor considered here, and shown in figure 6.1, the schedule becomes very simple: **{W, M, E, D, F}**, implying that in each cycle, the transitions are evaluated according to the specified order.

The quasi-static schedule implies that the search for enabled binding elements is done much faster because it is now a question of evaluating the transitions in the order specified by the schedule as opposed to regular HCPN simulation where such a schedule cannot be determined prior to runtime and hence the schedule of fireable transitions needs to be computed at runtime in each simulation cycle. Secondly, the quasi-static schedule based on the topological sorting of the transitions implies that when a transition is evaluated and it is found to be enabled, it can directly produce the tokens specified because the transition, which have the output places connected to the current transition as input places, will already have been evaluated. This means that the enablement of that transition will not be evaluated until the next simulation cycle in order to ensure correct synchronous behaviour. In traditional HCPN engines, it is necessary to use a two phase algorithm which first evaluates all enabled transitions and consumes the specified input tokens and then, when all transitions have consumed their

input tokens, a second sweep is made on all the transitions that were allowed to fire in order to produce their specified tokens. Only in the case where feedback arcs exist, or when a transition is connected to an output place of an interface, is it necessary to use the more costly two phase evaluation principle in order to ensure correct operation in the modified HCPN based model-of-computation presented here.

Another important optimization, which can be performed in the presented model-of-computation and which cannot be performed in generic HCPN simulations, is based on the observation that only two types of tokens are used, one representing services, and a second representing service requests. Service tokens are used solely to model the availability of resources and thus can be seen as passive tokens not being able to enable a transition alone. Service request tokens, on the other hand, are active tokens and are the only type of tokens which can enable a transition implying that, when evaluating the enablement of a transition, only the input places of the transition which contains service request tokens need to be checked. However, service tokens play an important role in the modelling of access to resources because the service model implementation can specify that their presence is needed before a given transition can be enabled - even though an active service request token is present.

Because the service model structure does not change during simulation, the possible routes of the service request can be determined before runtime and, thus, it is possible to extract information regarding all the transitions and places of the model through which the service request can pass. This information is used to create a service description which captures the behaviour of the service in the individual parts of the model for each service offered by a HCPN based service model. The service request description contains one member function for each place which the service request can pass through in the model and the service request is instantiated with references to all the variable objects which are required for reading or writing during the execution of the service request. The place member functions capture the functionality associated with firing one of the desired output transitions of the particular place and are built around a simple template containing the following elements:

1. Evaluate transition guard.
2. Evaluate destination place guard.
3. Check for availability of required services / resources.
4. Transition action.
5. Produce output tokens.
6. Consume input tokens.

In this way, only the functionality relating to the specific service request is

evaluated and can be executed directly. In the case of the ARM9TDMI processor there is only one possible path for all service requests. However, different paths could easily exist, as will be illustrated later in this chapter.

A concrete example of the use of place member functions for capturing the detailed behaviour of the service requests of the ARM9TDMI processor core is shown in listing 6.1 and 6.2, where a place member function is defined for each place through which the service request can pass. When the execute method is called, it will model the firing of the individual pipeline stages of the service model of the ARM9TDMI core by calling the place member functions one by one in the order specified by the quasi-static schedule.

The abstract service request class described in listing 6.1 and 6.2 defines the basic behaviour of all service requests of the ARM9TDMI service model and is extended by the child classes which capture the actual implementations of the individual service requests. Thereby, these will then specify the specific functionality. In this way, the functionality of each pipeline stage can be captured at a level of detail which is determined by the designer of the model. In listing 6.1 it can e.g. be seen how a call to the place member function **sr** first checks the guard expression and then models the firing of transition **F**, increments the program counter before the implementation specific functionality of the child class is allowed to occur and, finally, the service request is moved to place **p1** and the next service request is fetched. Pipeline-interlocks are modelled, as can be seen in the place member functions **p1** and **p3** of listing 6.1 and 6.2, using guard expressions and masks, which determine if the current service request requires reading of register operands which are to be modified by a write further down the pipeline. If this is the case, the pipeline is stalled because the guard expressions will then evaluate to false.

As described in section 3, the service model concept uses the notion of processes to model concurrent activity. Service models described using the HCPN based model-of-computation use only a single process to capture the behaviour of the synchronous component being modelled. The inherent parallelism of the component is then captured using transitions. The quasi-static schedule extracted from the model before runtime is used to construct an execute method which is called every time the process of the model is activated by the simulation engine. The process is activated periodically according to a specific clock frequency specified by the designer of the model. Simulation is then carried out by evaluating the places in the order determined by the quasi-static schedule specified in the execute method. To illustrate this, the execute method of the ARM9TDMI processor model is shown in listing 6.3. Due to the quasi-static scheduling and the member functions extracted for all service requests tokens, it is simply a matter of calling the member functions of the tokens in the order specified by the schedule. The member functions will evaluate any guard-expressions, perform the transition actions as well as handling the consumption and production

```
abstract class ARM9TDMIServiceRequest {
  ...
  @Override
  public final void sr() {
    // Guard expression
    if (!p1.hasServiceRequests()) {
      // Default action
      fProgramCounter.increment(4);

      // Implementation specific fetch action
      this.F();

      // Produce
      p0.addServiceRequest(this);

      // Consume
      sr.removeServiceRequest(this);

      // Fetch next service request
      sr.addServiceRequest(fProgramMemory[
        fProgramCounter.getIntegerValue()]);
    }
  }

  @Override
  public final void p0() {
    // Guard expression
    if (!p1.hasServiceRequests()) {
      // Check reservation mask - pipeline inter-locks.
      if ((WRITE_MASK & REGISTER_READ_MASK) == 0) {
        // Implementation specific decode action
        this.D();

        // Update reservation mask - lock destination register
        WRITE_MASK |= REGISTER_WRITE_MASK;

        // Produce
        p1.addServiceRequest(this);

        // Consume
        p0.removeServiceRequest(this);
      }
    }
  }
```

Listing 6.1: First part of the base service request implementation of the ARM9TDMI processor core.

of tokens, e.g. moving itself to the new place.

To summarize, the result of the optimizations described in this section is a simulation engine in which it is possible to obtain the flexibility of interpreted

```
    @Override
    public final void p2() {
      // Guard expression
      if (!p2.hasServiceRequests()) {
        // Implementation specific execute action
        this.E();

        // Produce
        p2.addServiceRequest(this);

        // Consume
        p1.removeServiceRequest(this);
      }
    }

    @Override
    public final void p3() {
      // Guard expression
      if (!p3.hasServiceRequests()) {
        // Implementation specific memory action
        this.M();

        // Produce
        p3.addServiceRequest(this);

        // Consume
        p2.removeServiceRequest(this);
      }
    }

    @Override
    public final void p4() {
      // No destination guard

      // Implementation specific write back action
      this.W();

      // Update reservation mask - release register
      WRITE_MASK &= ~REGISTER_WRITE_MASK;

      // Consume
      p3.removeServiceRequest(this);

      // Signal done
      this.done();
    }
  }
```

Listing 6.2: Second part of the data-processing instruction implementation of the ARM9TDMI processor core.

```
/**
* The main execution method of the ARM9TDMI model.
*/
final IBlock fB0 = new IBlock() {
  @Override
  public final IBlock execute() {
    if (p3.hasServiceRequests()) {
      ((IARM9ServiceRequest) p4.getServiceRequest()).p3();
    }
    if (p2.hasServiceRequests()) {
      ((IARM9ServiceRequest) p3.getServiceRequest()).p2();
    }
    if (p1.hasServiceRequests()) {
      ((IARM9ServiceRequest) p2.getServiceRequest()).p1();
    }
    if (p0.hasServiceRequests()) {
      ((IARM9ServiceRequest) p1.getServiceRequest()).p0();
    }
    if (sr.hasServiceRequests()) {
      ((IARM9ServiceRequest) sr.getServiceRequest()).sr();
    }

    return fB0;
  }
```

Listing 6.3: The execute method of the service model of the ARM9TDMI processor.

simulations and, at the same time, some of the speed obtained using compiled simulations. However, in this approach it is not the application running on the models (in our cases represented by service requests) which are compiled; it is only the service model description corresponding to the approaches taken by instruction set compiled simulations.

6.3 Experimental Results

In this section, simulations using the ARM9TDMI processor service model described above, as well as service models of an ARM7TDMI and an XSCALE processor core will be presented. The ARM7TDMI core has a 3-stage pipeline, and thus is simpler than the ARM9TDMI core, whereas the XSCALE processor has a relative complex pipeline.

The ARM7TDMI and ARM9TDMI processors implement the ARMv4T instruction set, whereas the XSCALE processor implements the ARMv5T instruction. The ARMv5T instruction set is a super set of the ARMv4T instructions and thus the processors can execute the same binary program images as long as pro-

grams are compiled targeting the ARMv4T instruction set. Simulation results from the three service models will be presented, primarily in order to illustrate the obtainable simulation speed as well as give more details on the modelling capabilities of the HCPN based model-of-computation.

The three service models have been implemented based on the information provided in the ARM reference manuals [66, 65, 67, 42]. Unfortunately, no cycle accurate instruction set simulators has been available for verification of the timing accuracy of the service models. Thus, the models have not been verified with respect to cycle accuracy. Instead only a functional verification has been made using the functional arm-simulator which is supplied with the GNUARM Tools [8] as reference. However, as the three service models capture each pipeline stage of the corresponding processor cores and use actions associated with each transition, representing the functionality of each pipeline stage, it should only be a matter of refinement in order to obtain service models which would be cycle accurate. To justify the correctness of such a statement, in chapter 7, a bit true and cycle accurate service model of a proprietary application specific processor from the portfolio of Bang & Olufsen ICEpower is described. In this case the full RTL description has been available and thus it has been possible to verify that the service model of the application specific processor is cycle accurate and bit true.

The three models implement the data-processing instructions, the branch instruction as well as the majority of the load and store instructions of the ARMv4T instruction set. The individual ARM instructions are relative complex and each instruction has several different variants depending on the parameters they are instantiated with. Also, all instructions are executed conditionally although the majority use a conditional expression which always evaluates to true. The implemented instructions allow for implementing the control flow of the applications using various branch, compare and test instructions, integer arithmetic such as subtraction, addition and multiplication as well as load and store instructions. The set of instructions modelled allows applications specified as C source code to be compiled using the **gcc-arm-elf** compiler tool-chain [1] to be executed.

The service models of the ARM7TDMI and the XSCALE processors were constructed in a way similar to the ARM9TDMI core described in the previous sections. The ARM7TDMI core is very similar to the ARM9TDMI core but has a simpler 3-stage pipeline with a fetch, a decode and an execute stage only. In the fetch stage, instructions are fetched from the instruction memory. In the decode stage, instructions are decoded and finally in the execute stage, operands are fetched, the instruction is executed and the result is written back to the destination. Figure 6.2 shows the graphical representation of the service model of the ARM7TDMI core.

In the case of the service model of the ARM7TDMI processor, the quasi-static

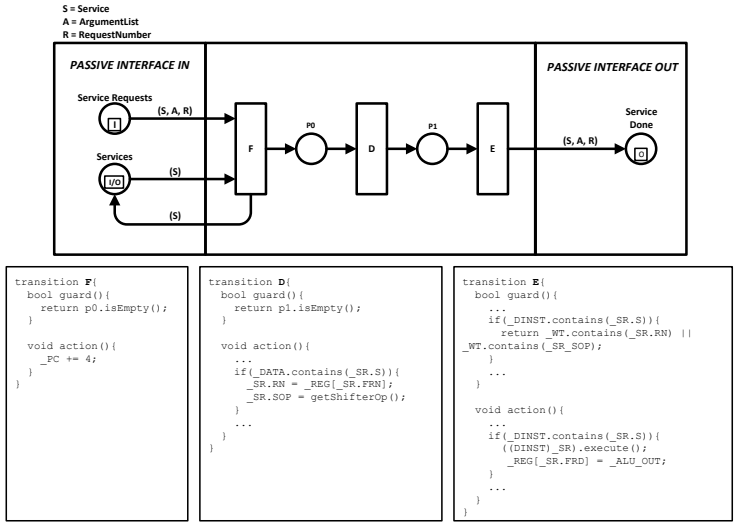


Figure 6.2: HCPN model of the ARM7TDMI processor which is an implementation of the ARMv4T instruction set.

schedule used to create the execute method of the model becomes very simple: **{E, D, F}**, implying that in each cycle, the transitions are evaluated according to this order. The service model of the ARM7TDMI core can be seen as a scaled down version of the ARM9TDMI model.

The pipeline of the ARM7TDMI and ARM9TDMI processor cores are relatively simple and, thus, the structure of the two service models described using the proposed HCPN based model-of-computation is, too. Thus, the more complex pipeline found in the XSCALE processor has also been modelled. Again, due to lack of a reference simulator, the service model of the XSCALE processor has not been verified for cycle accuracy. However, the full pipeline is modelled and pipeline inter-locks, caused by data dependencies, are modelled in the same ways as was done for the ARM7TDMI and ARM9TDMI processor service model as well.

The XSCALE is an in-order-issue-out-of-order-completion processor, having a shared fetch and decode pipeline and then parallel memory, execute and multiplication pipelines. Figure 6.3 shows the structure of the service model capturing the XSCALE processor core. Again, actions are associated with the transitions of the model in order to capture the behaviour of the individual pipeline stages. Again, each instruction is modelled using a service request, and the information regarding the possible paths of the individual service requests can be extracted at compile time allowing the possible routes to be encoded in the individual service request descriptions.

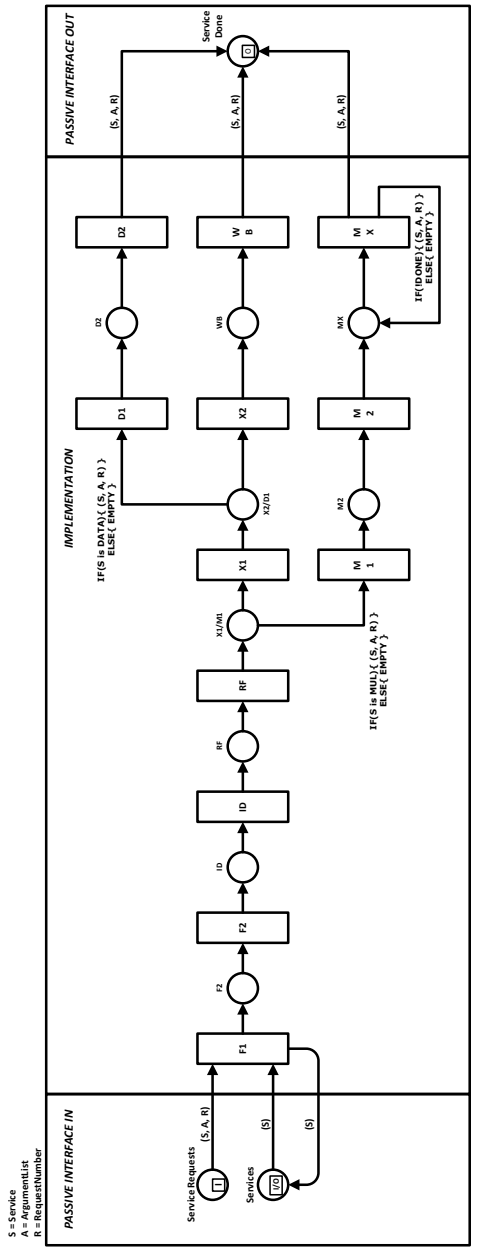


Figure 6.3: HCPN model of the Intel XScale processor which is an implementation of the ARMv5T instruction set.

Previously, it was shown how a single base class was used to represent all service requests of the ARM9TDMI core because only one possible route for all service requests was possible. In the case of the XSCALE service model, three different possible routes can be taken; hence, three different base classes are defined:

1. Load-store instructions which will take the following route:
   SR– >**F2**– >**ID**– >**RF**– >**X1**– >**X2**– >**D2**

2. Data-processing instructions which will take the following route:
   SR– >**F2**– >**ID**– >**RF**– >**X1**– >**X2**– >**WB**

3. Multiplication instructions which will take the following route:
   SR– >**F2**– >**ID**– >**RF**– >**X1**– >**M2**– >**MX**

Similarly, the transition firing order can be extracted as well, forming the quasi-static schedule used to construct the execute method of the XSCALE service model which is shown in listing 6.4.

Returning to the simulations performed using the three different ARM service models, a number of algorithms were used for investigating the obtainable simulation speed of the a three different ARM service models which can be categorized as bit true and roughly cycle accurate. In order not to avoid a correlation between the obtainable simulation speed and the particular application, a number of different applications have been investigated. However, in the implementation of the service models, only basic integer arithmetic is implemented and, thus, all applications are written in such a way that they used fixed-point arithmetic. Five different simple applications were chosen which avoid system calls and other non-supported function calls. These were: A cyclic-redundancy-check (CRC) application, a Fast-Fourier-transform (FFT) application, a Inverse-Fast-Fourier-transform (IFFT) application, and a finite-impulse-response (FIR) application and finally a loop-test using four nested for-loops. The binary images were generated using the **gcc-arm-elf** [1] compiler with ***no*** optimizations enabled.

Simulations were then carried out using the presented system level modelling and performance estimation framework. A simple platform model was constructed composed of a single service model only. The platform model was then configured to use the ARM7TDMI, ARM9TDMI or the XSCALE service model and configured with separate program and data memory each having a size of 16K words. A system model was then formed by specifying the binary image of the application which was to be executed on the instantiated service model. The functional results were then compared and verified with the functional ARMv4T simulator [1] and table 6.1 summarizes the results. It should be noted that all simulations are performed on an Intel Core 2 Duo T7200 processor running 2.0 GHz and with 2GB RAM.

```
/**
* The main execution method of the XSCALE model.
*/
final IBlock fB0 = new IBlock() {
  /*
   * (non-Javadoc)
   *
   * @see com.sysmopee.core.model.process.IBlock#execute()
   */
  @Override
  public final IBlock execute() {
    if (d2.hasServiceRequests()) {
      ((IXSCALEMemoryServiceRequest) d2.getServiceRequest()).d2();
    }
    if (d1.hasServiceRequests()) {
      ((IXSCALEMemoryServiceRequest) d1.getServiceRequest()).d1();
    }
    if (mx.hasServiceRequests()) {
      ((IXSCALEMulServiceRequest) mx.getServiceRequest()).mx();
    }
    if (m2.hasServiceRequests()) {
      ((IXSCALEMulServiceRequest) m2.getServiceRequest()).m2();
    }

    if (wb.hasServiceRequests()) {
      ((IXSCALEDataServiceRequest) wb.getServiceRequest()).wb();
    }
    if (x2.hasServiceRequests()) {
      ((IXSCALEDataServiceRequest) x2.getServiceRequest()).x2();
    }
    if (x1.hasServiceRequests()) {
      ((IXSCALEDataServiceRequest) x1.getServiceRequest()).x1();
    }
    if (rf.hasServiceRequests()) {
      ((IXSCALEServiceRequest) rf.getServiceRequest()).rf();
    }
    if (id.hasServiceRequests()) {
      ((IXSCALEServiceRequest) id.getServiceRequest()).id();
    }
    if (f2.hasServiceRequests()) {
      ((IXSCALEServiceRequest) f2.getServiceRequest()).f2();
    }
    if (sr.hasServiceRequests()) {
      ((IXSCALEServiceRequest) sr.getServiceRequest()).sr();
    }
    return fB0;
  }
};
```

Listing 6.4: The execute method of the service model of the XSCALE processor.

The results show that reasonably fast simulation speeds are obtainable using the

	ARM7TDMI	ARM9TDMI	XSCALE
CRC	9.7M	8.8	5.1M
FFT	7.6M	5.0	4.7M
IFFT	7.5M	4.9	4.1M
FIR-1024	7.5M	7.0	4.3M
Loop-test	7.9M	7.1	4.4M

Table 6.1: Obtainable simulation speed of the three different ARM models (simulation cycles / second).

service models of the three different ARM processors in which the individual pipeline stages are modelled explicitly. The service models of the ARM7TDMI and the ARM9TDMI core are believed to be very close to cycle accurate whereas the XSCALE service model is probably further from being cycle accurate as advanced features, such as branch target buffers, are not modelled in the current version.

The obtainable simulation speeds are promising but might degrade slightly in the process of refining the models to be truly cycle accurate. The goal of the HCPN based model-of-computation is to be adequately fast and at the same time allowing models using the HCPN based model-of-computation to be used in the system level modelling and performance estimation framework presented in this thesis and, thus, the current obtainable simulation speeds are definitely very promising.

The service models introduced here model only the core of the processors described. However, the memory controllers, caches, memories, co-processors and busses could be modelled using HCPN based service models in a similar matter if desired.

6.4 Summary

In this chapter, a model-of-computation was presented based on Hierarchical Coloured Petri Nets (HCPNs) which allow the generation of quantitative performance estimates of synchronous hardware components through simulation. The model-of-computation allows models described at different levels of abstraction to co-exist, ranging from high level functional models to detailed cycle accurate and possibly bit true descriptions, while being much faster than traditional register transfer level simulations.

The time required to describe models using the presented framework is much less than the time required to describe the equivalent RTL model due to the higher level of abstraction used. It is significantly faster to modify a model

described using the HCPN based model-of-computation in case of bug fixes or functionality extensions. This is especially beneficial in the development of new components where the exact requirements are not yet known. In such a case, the framework can be used for the architectural exploration of the component and then followed by an RTL description in a hardware description language such as Verilog, etc. Currently, it is not possible to automate the generation of RTL descriptions. However, due to the information included in the case where a detailed bit true and cycle accurate model exists, it is believed that in the future this can be done at least partially automated.

Furthermore, in the category of future work, an important and very interesting path will be the investigation of the possibilities of using the existing formal analysis capabilities defined for HCPNs. It would be very nice to be able to couple the promising simulation efficiency shown with the possibilities of formal analysis in order to formally reason about the properties of a model before simulations are performed.

Chapter 7

Exploration of a Digital Audio Processing Platform

In this chapter, the presented system level performance estimation framework is applied to a non-trivial industrial case provided by the Danish company and DaNES partner Bang & Olufsen ICEpower. The case study illustrates how the framework can be used for quantitative performance estimation using a simulation based approach and a successive refinement of models. The framework is applied in a top-down approach using an iterative refinement process that allows qualified decisions to be made at each level of iteration based on quantitative feedback extracted from the framework.

In this case study, a mobile audio processing platform is considered. The purpose of the case study is to illustrate the potential of the presented framework and to give an insight into its usage. As already mentioned in chapter 1, Bang & Olufsen ICEpower is working within the field of audio power conversion. Currently, an existing version of the audio processing platform considered is available on the market and is sold in high volumes. However, due to a very competitive market, the platform is constantly being refined in order to reduce cost and increase performance and/or functionality.

The mobile audio processing platform, illustrated in figure 7.1, is comprised of a digital front-end and a class D amplifier including the analogue power stage on-chip. The platform offers stereo speaker and stereo headphone audio processing resulting in a total of four audio channels being processed.

The digital front-end includes audio and control interfaces, sample rate conver-

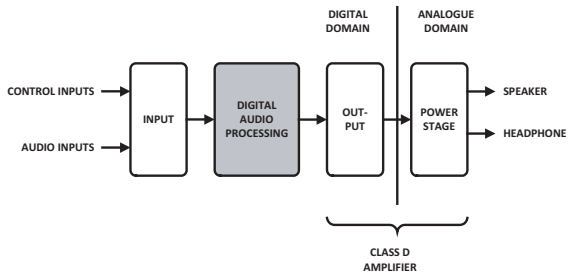


Figure 7.1: Overview of the case-study platform. The shaded block is the focus of this case-study.

sion, mixing, audio processing algorithms for channel enhancement and finally a proprietary digital modulator from which the output is connected to the on-chip analogue power stage. However, in this case study, the focus will be on the digital audio processing part of the platform only in which all audio processing is done within a single clock domain. The input interfaces, mixing and sample rate conversion are considered one combined abstract entity from which audio samples are being sourced at a fixed rate determined by the sample rate of the system. Similarly, one processed sample must be available at the modulator at each sample rate period in order to ensure normal operation. The ratio, defined in equation (7.1), between the sample rate of the system and the clock-frequency of the audio processing clock domain, expresses the real-time constraint of the application, i.e. how many cycles that can be used for audio processing per sample.

$$Cycles/sample = \frac{F_{clk}}{F_{samplerate}} \tag{7.1}$$

The sample rate of the system is defined as part of the specification of the application and cannot be changed but the real time constraint can still be modified by changing the clock frequency of the audio processing domain in order to scale the number of available cycles for processing each sample. However, with the targeted domain of the platform in mind, the objective is to have the lowest possible clock frequency in order to minimize the power consumption. The real time constraint can also be modified using buffers so that the processing of a buffer can amortize any processing overhead on the complete buffer limited, both as regards the overall tolerated latency of the application, and especially, in the current case-study, as regards the physical area occupied by the buffers.

The audio processing application under investigation can obviously be categorized as a stream based application. However, the application also includes the possibility of customizing and controlling the audio processing at runtime

adding a less dominant control oriented nature to the application. This allows the individual processing algorithms to be turned on and off as well as changing the operation of each one through the change of various parameters. Some of these control operations are time constrained implying that they must be handled within a given time frame in order not to have a negative impact on the subjective listening experience.

The objective of the exploration is to optimize the execution platform, on which the application runs, in terms of silicon area and power consumption, both of which need to be minimized. Due to the very high production volumes of the system, both the cost and performance of the system are critical elements in order to obtain commercial success, making the flexibility of the implementation a secondary objective only. In addition to these objectives, the time-to-market constraints of the system is added, complicating the design process even further by limiting the possibility of exploring the design space severely due to the absence of the possibility of getting feedback on design choices until the system has been realized at a very low level of abstraction with the current tools available. The increasing time-to-market constraints, however, are making a flexible implementation more attractive both in terms of design time for the current design but also increase the possibility of reusing the platform in future systems.

Traditionally, the choice of implementation has been between a platform based on a DSP processor or a fully dedicated hardware implementation. The DSP processor is programmable and includes a software debugging environment, making algorithm implementation and debugging relatively easy compared to a hardwired implementation done at register transfer level. However, the efficiency of the DSP compared to the hardwired implementation in terms of area and power consumption is sacrificed to some degree in exchange for flexibility which implies that a DSP based platform at first glance seems uninteresting, the current objectives taken into consideration. It seems obvious that both solutions have severe drawbacks either with respect to efficiency or with respect to the level of flexibility. Thus, a third option has been considered at the company and an experimental application specific processor referred to as the *SVF*-processor has been developed. The SVF processor is optimized to execute the type of algorithms needed in the application of the current case-study, has a relatively small silicon footprint, a very shallow pipeline and is programmable by offering 61 different instructions. The rather general DSP includes a number of features which are not needed by the specific application considered. Thus, a platform based on one or more SVF processors might prove to be the better choice in the trade-off between flexibility and efficiency but still need not be efficient enough to compete with the hardwired implementation.

The major problem with the three types of solution is that even though a number of conclusions seem obvious based on experience and intuition, they cannot be

verified until a very late stage in the design process where, as a minimum, register transfer level descriptions of the systems exist and, so, answering the question, *"What is the best suited platform for the given application under the given constraints?"* is not straightforward in the early design phases.

In the following, it will be explained how the presented framework was used in order to help answer the question before the system is realized at a very low level of abstraction based on quantitative performance estimates produced by the framework. The accuracy of the estimates, as well as the simulation speed, will also be discussed.

7.1 Application modelling

The first step in order to start the exploration of the platform, using the proposed framework, is to construct an application model. The application model is constructed from a specification of the application in Matlab and captures the functional behaviour of the application in a number of tasks as well as specifies the communication requirements of the individual tasks explicitly, without any assumptions on the implementation, following the principle, on which the framework is founded, of separating the specification of functionality, communication, cost and implementation.

The audio processing consists of four different algorithms which are labelled **A**-**D**. Table 7.1 shows their computational requirements in terms of arithmetic operations per sample only.

Algorithm	Number of Multiplications	Number of Additions / Subtractions
A	15	15
B	8	6
C	3	2
D	3	12
Total	29	43

Table 7.1: Table showing the computational requirements of the audio processing algorithms in terms of aritmetic operations *per sample*.

The application supports the processing of a total of four audio channels allowing individual stereo speaker and stereo headphone processing. The application receives one common stereo audio stream, consisting of a left and right audio stream. After the processing of algorithm **A**, these are split into four separate audio streams, two for speaker and two for headphone, in order to allow a separate processing of the speaker and the headphone streams. The resulting high level application model is shown in figure 7.2.

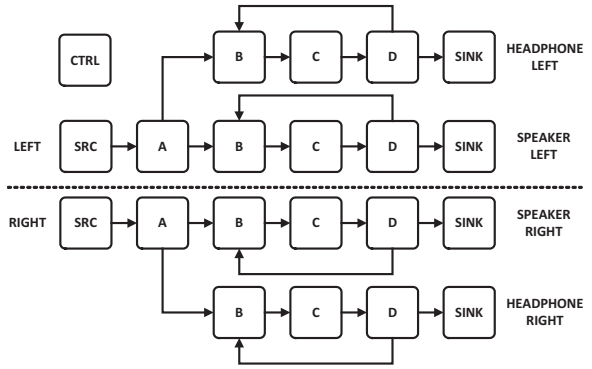
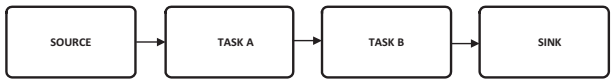


Figure 7.2: High-level view of the application model of the case-study.

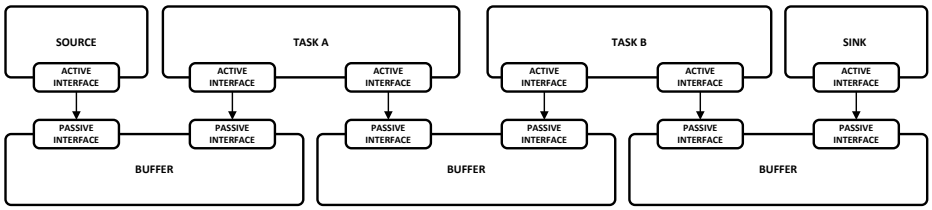
The application model contains 14 tasks (2 x A, 4 x B, 4 x C, 4 x D) for modelling the processing of the audio streams, 6 tasks (2 x SRC, 4 x SINK) for modelling the audio interfaces to and from the environment, as well as the environment itself, and finally an additional task (CTRL) for modelling the changes of the application state and/or control parameter changes which influence the individual audio processing parts. The connections between the tasks of figure 7.2 represent abstract communication buffers. The bandwidth of these are fairly low, each requiring the transfer of one audio sample in each sample rate period and with a word size which is implementation specific. The abstract communication buffers allow implementation independent inter-task communication and the implementation is considered only when the tasks are being mapped to the processing elements of the platform on which they run. In addition to the connections shown, all tasks have a connection to the control task allowing each one to access the parameters relevant to that particular task.

The tasks and buffers of the application model are modelled as a service models and in order to illustrate this in more detail, only a subset of the application model will be discussed in the following for illustrative purposes. This is done without loss of generality. Figure 7.3(a) shows a subset of the tasks of the application considered: A source task from which samples are generated, two audio processing tasks and a sink task to which processed audio samples are sinked. From the behavioural specification of the application in figure 7.3(a), an application model is constructed as shown in figure 7.3(b). The application model consists of one service model for each task and a buffer service model for each communication link required.

The functional behaviour of the application model was verified through simulations of the application model. The results were compared to the specification of the algorithms in Matlab and verified. The functional tasks of the high level application model are implemented using the floating-point arithmetic represen-



(a) The high level view of the simplified application model of the case-study.



(b) The coressponding simplified application model of the case-study, showing the individual service models and their inter-connection. For each task and each buffer, a service model is used to represent the functionality offered and required.

Figure 7.3: Application model construction.

tation offered by the host machine on which the simulator, running the model, is executed. This high level functional application model serves as the functional reference in the refinement steps towards the final implementation. However, at this level of abstraction, there is no notion of time or physical resources - hence only very rough performance estimates can be generated from a profiling of the application model combined with an annotation of the individual tasks with a cost. The cost can either be estimated or, in the case where the target platform is already available, a statistic average may be used.

7.2 Platform Modelling

In order to generate quantitative performance estimates, the tasks and buffers of the application model must be mapped to the components of a platform model creating a system model. Performance estimates relevant for evaluating the different platform options can then be extracted from the simulation of the system model.

When the tasks of an application model are mapped to the processing elements of a platform model, the tasks can, when executed, request the services offered by the processing elements. In this way, the functionality of the task is represented by an arbitrary number of requests to services which, when executed, model the execution of a particular operation or set of operations. The execution of a service can include the modelling of required resource accesses and latency only, or, depending on the level of abstraction used to describe the service model onto

which the tasks are executed, even include the actual functionality including bit true operations.

In the current case-study, three different types of platforms were considered based on either a DSP processor, the SVF processor or a dedicated hardware implementation. Due to area constraints of the system, the number of feasible platforms were limited to the platforms shown in table 7.2. The table also includes area estimates of the platforms relative to the dedicated hardware implementation. For each of the platforms listed in table 7.2, a platform model was constructed.

Platform	Description	Relative Area
HW	Dedicated hardware	1.0
SVFx1	1 SVF ASIP	0.6
SVFx2	2 SVF ASIP's	1.3
SVFx3	3 SVF ASIP's	2.1
SVFx4	4 SVF ASIP's	2.9
DSP	1 Audio DSP	4.0

Table 7.2: Relative area of the investigated platforms.

In the exploration of the platforms, the first set of quantitative performance estimates were focusing on an execution time analysis of the platforms only. At this point, the objective was to investigate the utilization of the processing elements of the platforms at different clock frequencies in order to find the lowest clock frequency at which the platforms could be run and still execute the application within the real-time constraint defined in equation 7.1.

The first step in the quantitative performance estimation process was to construct service models of the different types of processing elements for use in the specified platform models. In this case-study, the architecture of the processing elements was already fixed; hence a detailed latency based model could be constructed of each processing element using the HCPN based model-of-computation introduced in chatper 6.

For illustrative purposes the SVF processor will be used as, example in the following. The pipeline of the SVF processor is very shallow and illustrated in figure 7.4 which shows the HCPN based service model of the SVF processor.

The HCPN model of the SVF processor defines two service model interfaces: One passive and one active, as shown in figure 7.5. The passive interface allows application models to access the services of the SVF processor model, i.e. modelling the execution of the application on the processor. The active interface is used to support the connection of the SVF processor model to other service models e.g. a memory block. The services offered by a HCPN based service

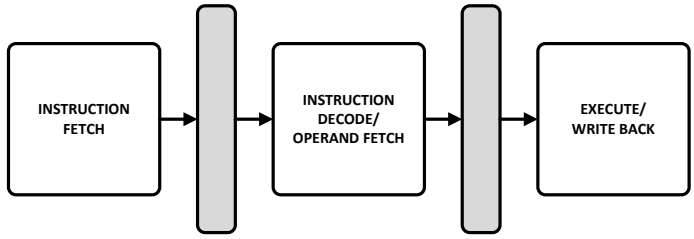


Figure 7.4: Simplified block diagram of the SVF-processor pipeline.

model are represented by a number of tokens in a one-to-one relation. In the case of the SVF processor model, each of the 61 instructions of the SVF processor is represented by a service and, initially, one token for each instruction is put in the place **Services**. During the simulation of a service model, a service is requested by placing a *service request* token in the place **Service Requests**. Recall, that a service request specifies the requested service, a list of arguments which can be empty, and a unique request number used to identify the service request by the simulation engine, e.g. to annotate the execution time of the service request. The argument list can be used to provide input arguments to the implementation of a service, or to allow the modelling of data operand dependencies, by letting the arguments specify one or more data operands that must be present before the service can be executed.

If the requested service is available in the **Services** place, and it is assumed that the service model is composed as shown in figure 7.5, the transition **T1** becomes enabled and is allowed to fire in the next simulation cycle. During a simulation cycle all concurrently enabled transitions are fired corresponding to the modelling of a global clock event. When the transition **T1** fires, the service request token from the **Service Requests** place and the corresponding service token from the **Services** place are consumed. The firing of the transition produces a new service token - of the type that was just consumed - in the **Services** place indicating that the model is ready for executing the same service again in the next simulation cycle. Furthermore, a service request token is produced in the second outgoing place of **T1**, having the same arguments as the service requests consumed including the unique request number which identifies the service request. The arrival of the service request in the **Service Done** place indicates the completion of the service request and the token representing the service request will be either consumed by the model or removed by the simulation engine for later analysis.

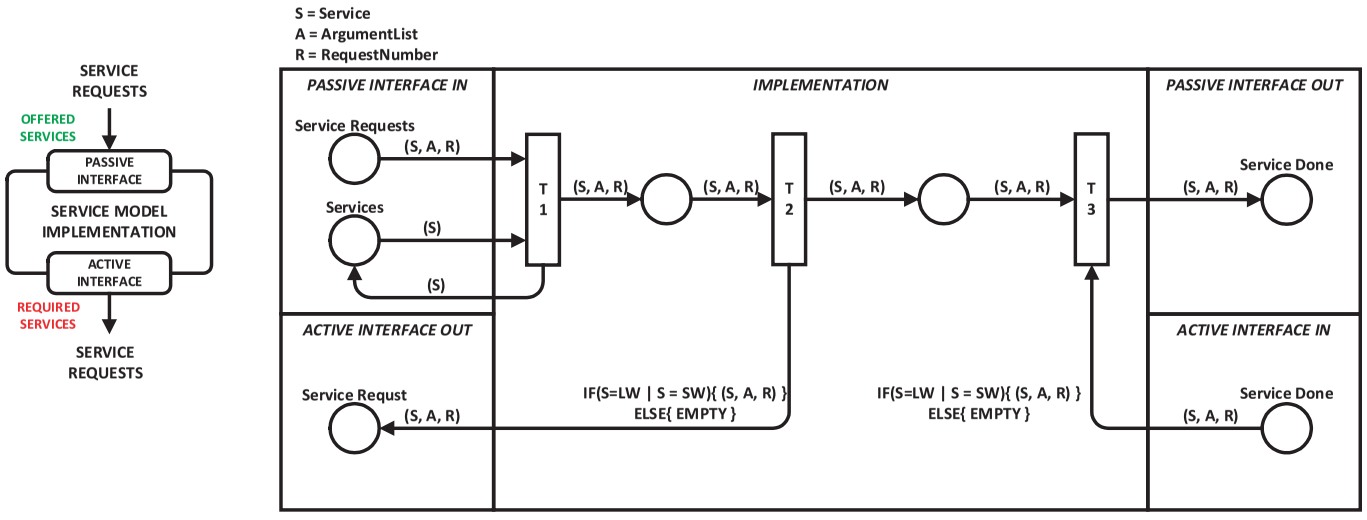


Figure 7.5: Service model of the 3-stage pipelined SVF application specific processor. The pipeline stages are clearly identifiable in the right hand side of the figure, separated by transitions. The model implements a single passive interface offering a service for each instruction of the processor and a single active interface requiring LW and SW services from an externally connected service model.

The service model implementation of the pipelined version of the SVF service model captures the latency of each service offered by the service model only. The pipeline stages are clearly identifiable in figure 7.5, separated by transitions. The model of the processor considered defines two interfaces: one, a passive service model interface through which the instructions, represented as a single service each, are offered to the application model which will eventually be mapped to the processor model, and a second, active, service model interface which allows the model to be connected to a memory model. The active service model interface also specifies the services which must be provided by the passive service model interface to which it will be connected. These, however, are not shown in the figure.

Arc expressions are used to route tokens through the model. When e.g. a load word (**LW**) or store word (**SW**) service is being processed, they request a new service request via the active service model interface of the SVF model of figure 7.5 modelling access to memory. Depending on the implementation of a service model, an arbitrary number of service requests can be processed in parallel e.g. modelling more advanced pipelines, VLIW, SIMD, and super scalar architectures.

If the framework had been used in the exploration of an architecture of a new processing element, the first version service model would probably have had an even higher level of abstraction, modelling a number of the key features very roughly only. The latency models do not include a modelling of the actual functionality but only the resource access and latency of each service without any modelling of data dependencies. This makes it possible to extract cycle approximate estimates of the execution of a given application.

7.3 Quantitative Performance Estimation

Quantitative performance estimation is then performed by mapping the application model to the platform models consisting of the latency based service models. The latency based service models do not model the actual functionality of the individual services and, so, no calculations are performed in the service models. This implies that the control flow of the application must be handled in the application model and the tasks mapped to the individual latency based service models must belong to the category of mixed-tasks. This group of system models is hence named **mixed latency**. The construction of a system model is illustrated in figure 7.6 where the application model shown in the upper part of the figure is being mapped to the **SVFx2**-platform model composed of only two SVF processor service models shown in the lower part of the figure. For reasons of clarity only part of the application model is shown in the figure. Similar system models were constructed for each of the platforms.

Figure 7.6: High level illustration of the **mixed latency SVFx2**-system model. For clarity only part of the application model is shown.

The utilization ratios extracted from the six different platforms are shown in figure 7.7 for the different clock frequencies investigated. In the cases where the application is requiring more computational cycles than those offered by the processing elements, implying that the real-time constraint of the application cannot be met, the utilization ratio has been set to 100% in the figure. Thus, in the figure, if a processing element has a utilization ratio of 100% it actually means that the platform is not usable. From the figure it can be seen that the clock frequency of the dedicated hardware implantation can be no lower than 25 MHz in order to fulfil the real-time constraint of the application.

The results of figure 7.7, combined with the area estimates of the platforms, showed that the SVF based platforms, were particularly interesting in the performance/flexibility trade-off having the possibility of achieving half the clock frequency of the hardwired implementation with only a minor increase in the silicon area in the case of the **SVFx2**-platform. The results also indicated that the clock frequency could even be reduced to one fourth of the hardwired implementation in the case of the **SVFx4**-platform at the expense, however, of a tripling of the area. The platform based on the general audio DSP proved not to be attractive due to a high area and medium performance compared to both the hardwired and application specific platforms.

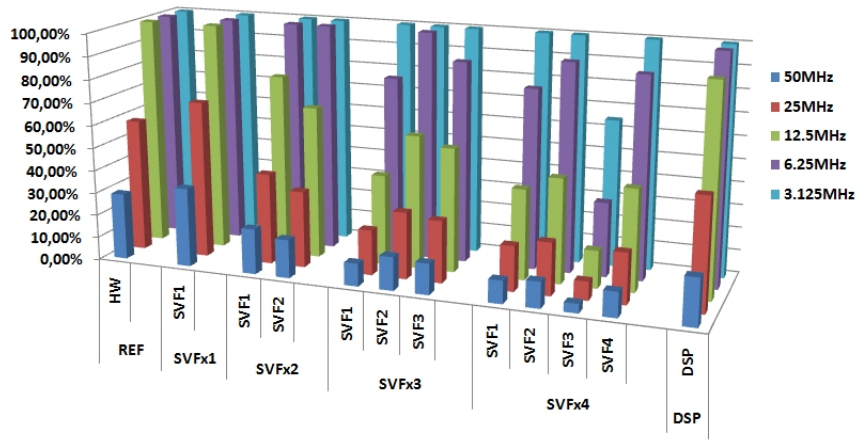


Figure 7.7: Platform utilization vs. clock-frequency.

Due to the promissing results obtained for the SVF based platforms these were selected for further modelling in order to verify the initial results. Therefore, a more detailed service model of the SVF processor were constructed and used for verifying the utilization ratios and the functional correctness of the results. The detailed SVF service model share the structure of the latency based model but

includes a detailed bit true modelling of the functionality of each service resulting in a bit true and cycle accurate model. The SVF processor is a synchronous 24-bit fixed point processor implying that data operations must be modelled explicitly due to the fact that the simulation platform used has a native word size of 32 bit. The source code for the detailed cycle accurate and bit true service model of the SVF processor can be found in appendix D.

The four SVF based platforms were refined to use the bit true and cycle accurate SVF service models utilizing the compositional properties of the framework and the tasks of the application models, which were mapped to the SVF service model processing elements, were now modelled as *implementation*-tasks. These system models are referred to as **compiled** system models, because the tasks are represented as a service request image generated using the existing compiler infrastructure associated with the SVF processor. In this way, a one-to-one correspondence with the physical execution of the application can be obtained. Furthermore, the platform models were refined to include service models representing FIFOs modelling direct point-to-point connections between the processing elements. Figure 7.8 shows the refined system model of the **SVFx2**-system. The platform model is still composed of two SVF processor service models, but now connected via a FIFO service model, allowing the left processor to send processed data samples to the right processor using a direct point-to-point connection. The figure also illustrates how the **SRC**-task is modelled as a *functional*-task in the application model and, still, it is allowed to communicate with the compiled **A**-task running on a cycle accurate and bit true model of an SVF processor through the abstract buffer **B1**. The possibility of mixing components described at different levels of abstraction is one of the strengths of the framework. Again, similar system models were constructed for each of the SVF based platforms.

The results of the simulations performed on the four compiled system models verified the functional correctness of these when compared with the results obtained from the application model. Furthermore, the results showed that it was actually the case that the **SVFx2**-platform makes it possible to lower the clock frequency to one half of the hardwired version. The results make the **SVFx2**-platform a promising alternative to a hardwired implementation when the area estimates are also taken into account. Even the **SVFx1** platform seems competitive with an area equivalent to the hardwired platform. However, this platform does not experience the benefit of halving the clock frequency as was the case with the **SVFx2** platform. Instead it has the same minimum clock frequency requirement as the hardwired version and, hence, it might be that the power consumption cannot compete with the hardwired platform. In order to reach such conclusions, a detailed analysis of the power consumption must be performed which is not possible in the presented framework.

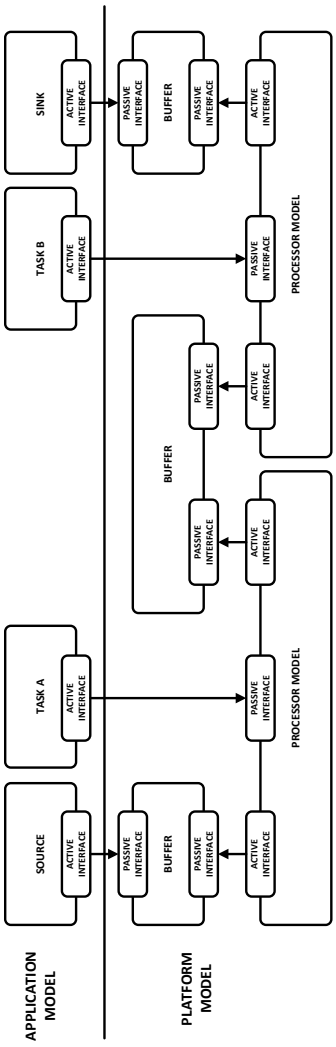


Figure 7.8: High level illustration of the detailed **compiled SVFx2**-system model. Again only part of the application model is shown for clarity.

7.4 Accuracy

The accuracy of the performance estimates produced using the framework depends directly on the level of abstraction used to describe the models used to generate the estimates. This implies that in order to use the performance estimates constructively, the level of abstraction at which the estimates have been generated must be taken into account.

In order to relate the quality of the performance estimates produced by the framework in the current case-study, an RTL implementation of the **SVFx1**-platform was created in the hardware description language Verilog referred to as the **RTL** model in the following. The simulations performed using the RTL model, were then compared with the results obtained from the performance estimation framework.

Table 7.3 shows the estimated number of cycles used to process a stereo audio channel produced by the framework for the **mixed latency** and the **compiled SVFx1**-system model and estimates extracted from the RTL model simulations.

The table shows that the cycle estimates obtained from the **mixed latency** model, in which only the latency based service models are used, are *not* cycle accurate. The cycle estimates produced by the **mixed latency** model are in general too optimistic and differ by roughly 18% for the **A** task, 2% for the **B** task and 9% for the **D** task. Only the estimated number of cycles for the **C** task is actually correct. This is caused by the fact that the latency based model does not take data dependencies into account. If desired, the modelling of data dependencies could be included and better estimates would be obtained, still using mixed task types. However, the purpose of these simulations was solely to get an initial rough performance estimate and, thus, this emphasizes the need for taking into account the level of abstraction with which the models are to be described when evaluating the results.

In the other range of the scale, in terms of accuracy, the table also shows the cycle estimates of the refined **compiled** model in which a cycle accurate and bit true modelling of the components was used. In this case, the cycle estimates are identical with the estimates obtained from the RTL model, as can be seen from the table, where the required computational cycles are listed for the processing of 3000 samples in one audio channel.

The constructed RTL model was also used to make a comparison of the functional results produced by the framework using the **compiled** version of the **SVFx1**-platform. Again, the comparison involved the processing of 3000 samples. The functional comparison showed that the audio streams processed from the system model were 100% identical to the processed audio streams from the RTL model.

	Mixed Latency	Compiled	RTL
A	84,034	102,034	102,034
B	123,041	129,384	129,384
C	33,011	33,011	33,011
D	153,051	168,056	168,056
Total	393,137	429,143	429,143

Table 7.3: Estimated number of cycles for the processing of 3000 samples in one audio channel at three levels of abstraction.

Furthermore, the time required to describe the model using the presented framework is much less than the time required to describe the equivalent RTL model due to the higher level of abstraction used. More importantly, it is significantly faster to modify a model of the framework in case of bug fixes or functionality extensions. This is especially beneficial in the development of new components the exact requirements of which are not yet known. In this case, the framework can be used for the architectural exploration of the component and then followed by an RTL description in a hardware description language such as Verilog, etc. Currently, it is not possible to automate the generation of RTL descriptions. Hopefully, however, this can be done at least partially automated in the case where a detailed bit true and cycle accurate model exists.

7.5 Simulation speed

Like the accuracy, the obtainable simulation speed also depends on the level of abstraction used to describe the components of the system model which is being simulated and the framework experiences the classic trade-off between accuracy and simulation speed. Table 7.4 shows the measured simulation speeds expressed as cycles per second for the individual system models investigated. Before proceeding, it should be noted that all simulations are performed on an Intel Core 2 Duo processor running 2.0 GHz and with 2GB RAM.

	Mixed Latency	Compiled	RTL
HW	21,9M	N/A	N/A
SVFx1	21,7M	19,9M	15,324
SVFx2	18,6M	15,8M	N/A
SVFx3	17,2M	13,8M	N/A
SVFx4	15,9M	12,4M	N/A
DSP	20,0M	N/A	N/A

Table 7.4: Obtainable simulation speed of the investigated system models (simulation cycles / second).

At the highest level of abstraction are the models referred to as **mixed latency** in which the application model tasks were modelled as mixed tasks and only the latency and resource access were modelled in the platform model. In general, these provide the fastest simulation speeds due to their high level of abstraction but, as mentioned in the previous sections, they can only be used for rough execution time analysis.

Table 7.4 also shows the obtainable simulation speeds for SVF-platform models which were refined to bit true and cycle accurate versions. Application models running on these are compiled into a service request image, equivalent to the actual binary image running on the actual hardware, using the compiler infrastructure of the SVF processor. These sets of simulations are referred to as **compiled** and only include the SVF-platforms.

However, what is more interesting is the big speed-up seen when comparing the simulation speed of detailed bit true and cycle accurate version of the **SVFx1**-system model with the equivalent RTL simulation. The **SVFx1**-system model runs at approximately 20 million cycles per second with all algorithms enabled, including data logging, and the functionally equivalent RTL description runs with approximately 15 thousand cycles per second resulting in a speed-up of more than 1000x. This makes it possible to use the bit true and cycle accurate system model as a virtual platform and also allows a much larger part of the functional design space to be tested and verified before a physical prototype of the system has been constructed.

7.6 Summary

In this chapter, a case-study from Bang & Olufsen ICEpower was presented in which a mobile audio processing platform was considered.

The results of the case-study indicate that the platform consisting of a single SVF processor is directly comparable to the hardwired solution in terms of gate count and processing power offered. However, what is more interesting, is that while the platform consisting of two SVF processors experience only a slight area overhead compared to the hardwired platform, it can actually run at half the clock frequency, potentially leading to a lower power consumption. At the same time, the SVF processors provide a higher degree of flexibility because they are programmable and, thus, design changes are adapted faster and also make a reuse of the platform more likely. However, the conclusion is not evident but, if the increased silicon area can be accepted, an SVF based platform is competitive with the current hardwired platform and may even lead to lower power consumption due to the lower clock frequency which can be used. Additional investigation is required, however, to draw such conclusions

due to the lack of power consumption estimates in the current version of the framework.

The application of the case-study was to be executed on a platform which was yet to be decided, and the company wanted to know how to ensure that the applications are executed in the best possible way, and which platform would be best suited for that. Usually, choices like these are based on the engineers' prior experience but using the presented framework, it was shown how to construct a model of both the application and of the platform that enables the two to interact and show how they will perform in a given situation within a relatively short time span. This allows a larger part of the design space to be explored, potentially leading to better designs and thus better products.

Part III

Extensions

Chapter 8

Service Model Description Language

This part presents initial work on two different aspects of supporting a practical use of the framework at companies like Bang & Olufsen ICEpower. Still, several areas of future work reside and pointers to these will be discussed in section 10.2.

The case study described in the previous chapter gave rise to a number of ideas for improvement of the system level modelling and performance estimation framework presented.

The first area of focus, which will be presented in this chapter, is the specification of synchronous hardware models using the presented HCPN based model-of-computation. In order to allow a specification of models which accommodate changes in an easier and more flexible way than the current one, in which the complete model is described in Java, a domain specific language referred to as the Service Model Description Language (SMDL) was developed. The primary purpose of having the current version of the SMDL language is to sketch a path to a usable approach which would allow an efficient practical use. For illustrating the basic features of the SMDL language, the experimental application specific SVF processor from Bang & Olufsen ICEpower, introduced in chapter 7, will be used.

A second area of focus, presented in chapter 9, is to allow automatic design space exploration (DSE) using the system level modelling and performance estimation framework described in this thesis to generate estimates of the performance of

the systems under investigation. The idea is to have a DSE framework which will automatically create permutations of the permissible configuration of a system model and then estimate the performance of the specific system instance through simulations. The current version of the DSE framework is based on the use of the NSGA-II heuristic for multi-objective optimization but, in principle, it should be very easy to use other heuristics if preferred. The DSE framework provides an infrastructure which allows designers to specify the objectives targeted as well as representing the decision variables and a number of evolutionary algorithm operators used for generating permutations when searching the design space. In order to illustrate the potential of the proposed framework, it will be applied with minor modifications to the case study which was explored manually in chapter 7.

The remaining part of this chapter introduces the initial version of a custom language for describing service models of synchronous hardware. The language can be categorized as an architecture description language (ADL). Currently, descriptions are used to generate fast simulation models using the service oriented HCPN based model-of-computation presented in chapter 6 only. However, in principle , the richness of the information contained in such descriptions makes it possible also to generate register transfer level descriptions in arbitrary hardware description languages, such as VHDL or Verilog. This will be discussed in the future work sections at the end of the chapter.

The chapter will start out with an introduction to architecture description languages and give an overview of the work related to the research within architecture description languages. The field of research has already been receiving much attention and several excellent description languages already exist. However, the nature of service models are so distinct that the flexibility of a custom language for this purpose outweighed the work required to develop this, compared to using a pre-existing architecture description language. Then, the language developed in this project will be described and examples will be given of how the language is used to describe the custom proprietary processor of the Danish company Bang & Olufsen ICEpower introduced in chapter 7.

8.1 Related Work

Architecture description languages are used to capture descriptions of architectures for many different reasons. The purpose of the description can be to generate compilers, simulators and possibly actual implementations of the described architecture in hardware through synthesis and automated test generation as illustrated in figure 8.1.

Architecture description languages can be categorized into three groups: Structural, behavioural or mixed structural and behavioural.

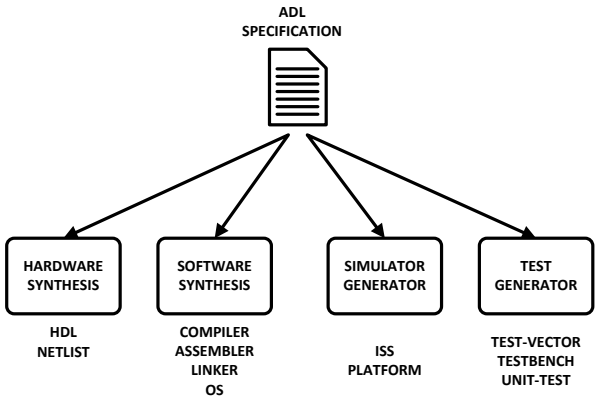


Figure 8.1: Usage of architecture description languages (ADL).

Structural ADLs focus on capturing the components of which an architecture is constructed and often include detailed descriptions of the actual hardware including functional units, pipelines etc. MIMOLA [61] is an example of a structural ADL in which the micro-architecture being modelled is captured through a net list of component models which are described at register transfer level as the behaviour of the component and a set of interfaces containing ports. Modules can be connected as seen in most HDLs. The information contained in the net lists is extracted by a code generator named MSSQ in order to generate the instruction set; this, however, is a very difficult task and constraints need to be inferred which basically limits the scope of targeted architectures.

Behavioural ADLs describe the behaviour of the operations offered only through a semantic description and, so, only implicitly describe the underlying hardware, leaving the task of extracting the hardware model to the software tools provided. In general, it is difficult to generate detailed cycle accurate models using behavioural based descriptions only without a number of assumptions, implying that the same problem exists if a synthesis of hardware is to be performed.

nML [31] is an example of a behavioural ADL which focuses on capturing the instruction set of a processor. nML has been used for several purposes including both automatic retargeting of compiler infrastructures and instruction set simulators. nML is also used commercially by Target Compiler Technologies and their tool suite CHESS/CHECKERS [98] for both compilation tool generation and instruction set simulator generation. nML uses a hierarchical specification of instructions in order to achieve a compact instruction set description. In this way common functionality is specified once only and shared among several instruction descriptions.

Several ADLs use a mixture of both the structural and behavioural elements and such an example is seen in [40] where EXPRESSION is introduced. EXPRESSION descriptions contain a behavioural part and a structural part. The behavioural part captures the instruction set whereas the structural part is used to capture the data path of the processor explicitly. The structural part includes timing information for elements which require multiple cycles, such as pipelined multipliers etc.

In many cases, the individual ADLs target a specific type of architectures, e.g. FACILE [94] which targets out-of-order processors, Sim-nML [41] which focuses on digital signal processors (DSPs). ISDL [39] is yet another behavioural ADL, and as nML is targeted at capturing instruction sets but targets primarily Very Long Instruction Word (VLIW) based processors and is mainly focused at automatic assembly and binary code generation.

LISA [108] is a very comprehensive example of an ADL which is also used commercially in the tool suites previously offered by CoWare and now, due to an acquisition, Synopsys [3]. Originally, LISA was developed for the same purpose as the SMDL language described in this chapter, namely for the generation simulators, but has subsequently evolved and is being used also for synthesis and compiler tool-chain generation.

Tensilica Instruction Extension (TIE) language [5] is an ADL developed by Tensilica [5] for describing application specific instruction set processors. The TIE ADL is used to describe the instruction set details including mnemonics, operands, encoding and the functional behaviour and timing requirements. Through software tools offered by Tensilica, an automatic retargeting of the GCC compiler tool chain [1] can be generated, as can custom instructions of configurable processor cores of the company.

The ArchC [10] which include software tools for automatically generating simulators and verification interfaces which allow refined models to be compared against the functional references and, thus, validating the correctness of the model.

TDL [55] focuses on generating post pass optimizers and analyzers for use in compilers based on machine-dependent information about the target processor captured using a specification in the TDL description language.

There has been extensive research within the field of architecture description languages for automatic simulator generation during the last two decades, as this section also shows, and only a very brief overview of a number of different ADLs have been presented here.

8.2 Service Model Description Language

In this section the details of the Service Model Description Language (SMDL) in its current form will be presented.

The SMDL architecture description language combines structural and behavioural descriptions in order to capture the synchronous hardware component being modelled. The aim of the current version, illustrated in figure 8.2, is to generate fast simulation models only and, consequently, it is not a full blown ADL as many of the ADLs discussed in the previous section. The SMDL descriptions are translated into service models implemented using the HCPN based model-of-computation described in chapter 6.

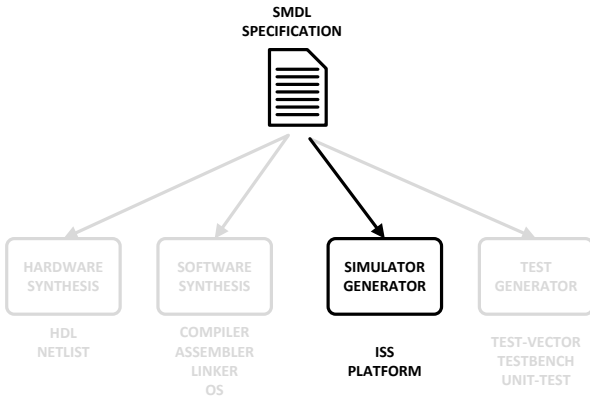


Figure 8.2: Usage of the service model description language - currently only simulation model generation is supported.

Structural descriptions are used for specifying the state holding elements of the component such as registers and memories. Currently, the actual model capturing e.g. the pipeline of a processor is created manually. However, it is only a matter of extending the SMDL language with syntax and semantics for capturing this as well; i.e. instead of specifying HCPN models in Java, it would be possible to do this in the SMDL language instead. However, this is considered a relatively trivial process because this is mainly a translation of a specification in one language to another. The time used to describe the services offered by a component is by far the most time consuming part of describing a service model, especially if bit-true modelling is needed, implying that non-native word-sizes are used to represent data, and if the data path bit widths have not yet been decided. Therefore, the focus is on the SMDL language in order to start with the possibility of capturing services.

The actual services offered by a service model described using the SMDL lan-

guage is specified behaviourally and can include the specification of access to structural elements in order to access e.g. data operands. The service description specifies the behaviour associated with the firing of the transitions of the structural HCPN model of relevance. In this way the service declaration of an SMDL description is used to automatically generate the member-functions of a service request as discussed in section 6.2.

Language elements

In order to introduce the developed language in detail, once again the application specific SVF processor developed at Bang & Olufsen ICEpower will be used as an illustration throughout this section. The complete SMDL source code for the SVF processor as well as the generated Java source code can be found in appendix C and D respectively. This section will outline the basic language elements of which the SMDL language is composed.

```
servicemodel SVF{
  /* Active interfaces */
  ActiveInterface SOURCE_LEFT;
  ...
  ActiveInterface SINK_LEFT;
  ...
  ActiveInterface I2C;

  /* Register file */
  word<24> reg[16];
  ...

  /* Program counter */
  word<16> pc;

  /* Service declarations */
  ...
}
```

Listing 8.1: Service model declaration.

A **servicemodel** declaration is used to capture the specification of a service model using the presented SMDL language. The service model declaration consists of a structural declaration and an arbitrary number of **service** declarations specifying the behaviour of the services offered by the model. An extract of the service model declaration of the SVF processor is shown in listing 8.1.

The structural declaration specifies the state holding elements of the model as well as the interfaces implemented by the model. In the future, the idea is that this structural declaration will also include the actual structure in terms of pipeline stages etc.

An **interface** declaration specifies the name and type of the interface, either active or passive. In the case of an active interface, the interface should specify the required services, i.e. the services which the model which implements the interface assumes to be requestable and which must be provided by the service model implementing the passive interface to which this active interface is connected. In the case of the declaration of a passive interface, the interface should similarly specify the services offered.

```
abstract service TYPE1 extends EXTENDEDTYPE2{
  encoding{
    OPCODE, OP1, OP2, OP3, EXT2, EXT1;
  }

  /* Pipeline behaviour */
  abstract void p0(){
    pc = pc + 1;
  }
  abstract void p1();
  abstract void p2();
}

service ADDS extends TYPE1{
  encoding{
    OPCODE = 1;
  }
  ...
  void p0(){
    super.p0();
    ....
  }

  void p1(){
    getOP1() = getOP2() + reg[OP3];
  }

  void p2(){
    ...
  }
  ...
}
```

Listing 8.2: Specification of the signed addition instruction of the SVF processor.

A **service** declaration, as illustrated in listing 8.2, is used to capture the behaviour of a service. The contents of listing 8.2, will be explained in details in the following sections.

A simplified object oriented structure is used allowing basic inheritance between services. Services can be declared abstract if the service is only intended to hold common functionality shared by extending child-services. Only the non-abstract service declarations will eventually be represented by a service request in the generated simulation model. Child-services can be declared as extending

a parent service, as is shown in listing 8.2, where the **ADDS** service is extending the abstract **TYPE1** service, in this way sharing the functionality declared by the parent service. This allows for a more compact description of services.

The body of a service declaration can specify an arbitrary number of **service method** declarations which are used to capture the behaviour of the service. The individual service methods can be associated with the execution of e.g. a pipeline stage and, so, the behaviour defined by the service method is executed when the corresponding transition of the HCPN model, modelling the functionality of the specific pipeline stage, when firing.

```
service ADDS extends TYPE1{
  encoding{
    OPCODE = 1;
  }
  ...
  void p0(){
    super.p0();
    ....
  }

  void p1(){
    getOP1() = getOP2() + reg[OP3];
  }

  void p2(){
    ...
  }
  ...
}
```

Listing 8.3: Specification of the signed addition instruction of the SVF processor.

Listing 8.3 shows the specification of three service methods. In this case, the method **p0** specifies the behaviour of the operand fetch stage of the **ADDS** instruction and, similarly, the method **p1** specifies the functionality of the execute stage and finally the method **p2** specifies the functionality of the write back stage of the pipeline of the SVF processor. The service method **p1** calls two methods, **getOP1** and **getOP2**, which are defined in a parent service. The two methods specify the operands to use, modelling the correct behaviour according to the corresponding instruction of the SVF processor. The variable **reg** is declared in the structural part of the service model declaration and is a global variable representing the register file of the SVF processor. The operand **OP3** is a constant specified by the encoding of the **ADDS** instruction declared in the abstract service **TYPE1**. The encoding declaration will be discussed later in this section.

A service method declaration can be declared abstract, but only if the service declaration itself is declared abstract. If a service method declaration is declared abstract it is up to the child service to implement the functionality of the

method. In listing 8.4, the abstract service **TYPE1** is declaring three abstract methods. The method **p0** defines common functionality by stating that the program counter variable should be incremented but leaves the specification of remaining functionality to be determined by any extending child services.

```
abstract service TYPE1 extends EXTENDEDTYPE2{
  abstract void p0(){
    pc = pc + 1;
  }
  abstract void p1();
  abstract void p2();
}
```

Listing 8.4: Illustration of the use of abstract services and methods.

Because the SMDL language is used extensively for describing processors, it also supports specification of instruction format encoding and can automatically generate a parser which converts binary encoded instructions to their corresponding service requests instances. This is done by including a special optional declaration in the service declaration referred to as an **encoding** declaration. In the encoding declaration the fields defined in the global encoding declaration of the service model, shown for the SVF processor in listing 8.5, can be referenced according to the actual desired encoding of the service. It is possible to include a binding of the global fields to specific values and/or force the presence of different number strings.

```
  /* Fields used in the encoding of the SVF Instruction Set */
  encoding{

    OPCODE[23:18]; OP1[17:14]; OP2[13:10]; OP3[9:6];
    IMMEEXT[11:0]; IMMEXT[9:3]; IMM[7:3];
    TARGET[17:0]; OFFSET[9:0];
    EEXT2[9]; EEXT1[8]; EXT2[5:3]; EXT1[2:0];

  }
```

Listing 8.5: Fields used in the encoding of the SVF Instruction Set.

The fields of listing 8.5 declare the position and number of bits used to encode the instruction set of the SVF processor which is shown in appendix B.

The encoding is determined by any parent service encoding declarations and the current service encoding declaration as illustrated in listing 8.6, where the abstract service **TYPE1** declares an encoding consisting of the fields **OPCODE, OP1, OP2, OP3, EXT2, EXT1**.

The **ADDS** service then binds the **OPCODE** field to the value of 1. The **ADDS** service can then be encoded as the binary string:

000001XXXXXXXXXXXXXXXXXX

where X represents arbitrary values. The binary string is used to generate a mask in the generated parser in order to instantiate the corresponding service from the binary application image obtained from the compiler infrastructure supplied with the SVF processor.

```
abstract service TYPE1 extends EXTENDEDTYPE2{
  encoding{
    OPCODE, OP1, OP2, OP3, EXT2, EXT1;
  }
  ...
}

service ADDS extends TYPE1{
  encoding{
    OPCODE = 1;
  }
  ...
}
```

Listing 8.6: The encoding of the **ADDS** service.

The part of the generated decoder responsible for instantiating the **ADDS** service request is shown in listing 8.7 which makes sure that all declared fields of the instruction are parsed as arguments to the constructor of the **ADDS** service request, too.

```
public IServiceRequest createServiceRequest(long inst) {
  ...
  else if ((inst & 262144) == 262144
    && (~inst & 16252928) == 16252928) {
    int l0 = (int) (inst & 960) >>> 6;
    int l1 = (int) (inst & 15360) >>> 10;
    int l2 = (int) (inst & 245760) >>> 14;
    int l3 = (int) (inst & 56) >>> 3;
    int l4 = (int) (inst & 7) >>> 0;
    return this.createServiceRequest("ADDS", new Object[] { l0, l1,
      l2, l3, l4 });
  }
  ...
}
```

Listing 8.7: The automatically generated binary decoder. The decoder decodes the binary instruction and returns an instance of the corresponding service request. This listing is an extract of the decoder for the **ADDS** instruction.

8.3 Service model generation

In this section, a brief overview is given of how simulation models are generated based on a specification of the model using the SMDL language. The current

implementation of the synthesis and code generation tool is built as a traditional compiler as illustrated in 8.3.

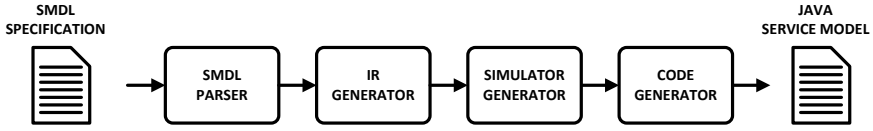


Figure 8.3: Usage of the service model description language - currently only simulation model generation is supported.

The SMDL specification is first parsed, checked for syntactical correctness and converted into an abstract syntax tree, in this way representing the syntactical elements of the specification as a data structure which allows the generation of an intermediate representation of the specification. A number of optimizations can then be applied to the intermediate representation. These are of a general nature and are independent of the optimizations which are implementation language specific. The optimized intermediate representation is then used to generate an abstract service model. This abstract service model is then parsed onto the code-generator which, in the current case, emits Java source code. In principle it should be possible to generate e.g. C/C++ or SystemC source instead in a straightforward manner at this point.

```
class ADDS extends TYPE1 {
  protected final IDataWord _FIELD0;
  protected final int _OP3;

  public ADDS(IServiceModel p0, IService p1, int OP3, int OP2,
   int OP1, int EXT2, int EXT1) {
   super(p0, p1, OP2, OP1, EXT2, EXT1);
   _OP3 = OP3;
   _FIELD0 = (IDataWord) fModel.getState().getRegister(
   "reg" + _OP3).getElement();
  }
  ...
  public final void p1() {
    _GETOP1.setIntegerValue((_GETOP2.getIntegerValue() + _FIELD0
    .getIntegerValue()));
  }
  ...
}
```

Listing 8.8: The Java implementation of the **ADDS** service request class which represents the ADDS instruction specified in SMDL.

Listing 8.8 shows the generated Java class for the **ADDS** service which were used as example in the previous section. All services specified in the SMDL specification will have a corresponding Java class.

In addition to the Java implementation of all services specified, a number of Java classes are generated automatically to support the service model concept using the presented framework. These make it possible for other models to request services and interface with the generated service model and also make it possible to e.g. allow the graphical user interface to inspect the variables declared, etc. Currently, the actual implementation of the service model, i.e. the part responsible for executing the service requests, must be specified manually. This, however, is the part which is the least time consuming to construct and it is estimated that this can be generated in the future with relative ease. The part of the HCPN based service model which requires the most time to specify are the actual services offered, especially if this is done at a low level of abstraction as in the case of the SVF processor example given here. The benefits of having a high level language, such as the SMDL language, is thus extra rewarding for auto generating the services of a model as can be done with the current version of the SMDL language and associated tools.

As can be seen in appendix C and D the SMDL description of the SVF service model is more compact than the equivalent specification of the SVF service model in Java. Also, the specification in SMDL is more intuitive and thus the SMDL specification brings several benefits seen from a practical point of view, allowing designers to specify service models of synchronous hardware components in a more convenient way.

8.4 Summary

This chapter presented the service model description language (SMDL) for specifying service models of synchronous hardware components in a fast and compact manner.

The SMDL language is still mostly used for proof-of-concept and hence much work still lies ahead within a further refinement of the language. It was shown how the SMDL language can be used through examples of how parts of the SVF processor were described. There are several elements which would be interesting to pursue in the future which follow traditional usage of ADLs. From a practical point of view, much value would be added if hardware synthesis support would be added as well as automated test generation in the form of both test benches and test vectors.

Until now the SMDL specification has only been used to generate fast simulation models based on the modified HCPN model-of-computation for modelling synchronous hardware components. Considerations regarding a future possibility of using the SMDL language for specifying all service models in general have also been made. The pros and cons of using a proprietary language for specification need to be evaluated. Pros, include the obtainable flexibility to match and allow

the specification of the concepts of service models easily. Heavy weighing cons include the complexity of managing a complete software tool-infrastructure as well issues regarding model interoperability.

Chapter 9

Automated Design Space Exploration

This chapter presents the first steps in the construction of an automated design space exploration framework which uses the system level modelling and performance estimation framework presented in part I for fitness evaluation of potential candidate solutions.

Design space exploration is the process of exploring the possible combinations of configurations of a system, quantifying each in order to associate a quality measure with the particular realization, allowing the best possible system realization to be selected. In theory, all possible design points need to be evaluated in order to find the optimal system. The problem in most cases is a multi-objective optimization problem in which several objectives are equally important implying that no unique solution exists. Instead of a single golden solution, a set of equally good solutions which represent the best trade-offs between the specified objectives can be identified from which the designers can then choose. The solutions which represent the best trade-off between the objectives are referred to as the *Pareto* optimal set of solutions.

In most practical instances, the sheer size of the design space is simply too large to allow a full evaluation of every design point. Thus, traditionally, the experience of the designers of a system has been a vital element in order to obtain a successful system realization. However, another approach is to use multi-objective optimization heuristics to facilitate a structured automated search of the design space and, so, limit the number of investigated points in the

design space in a structured manner. Such an approach, based on what is known as multi-objective evolutionary algorithms, is outlined in this chapter using one specific heuristic which, combined with the system level modelling and performance estimation framework presented earlier in this thesis, allows designers to perform an automated multi-objective design space exploration. The output of the proposed framework is a set of Pareto approximated solutions which express the best found trade-offs between the specified objectives from which the designer can then choose the preferred solution.

Figure 9.1 gives an overview of how the proposed design space exploration framework interacts with the system level modelling and performance estimation framework presented.

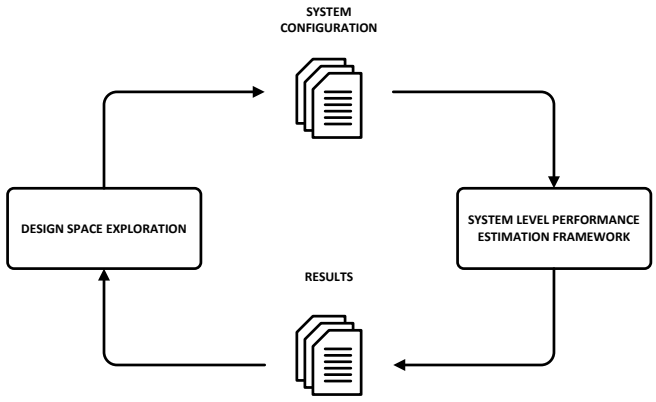


Figure 9.1: Overview of the proposed design space exploration framework.

In the remainder of this chapter, a short overview of related work will be presented before multi-objective optimization is briefly introduced. Then, one particularly well-known meta-heuristic for multi-objective optimization is introduced together with a number of operators used in the particular algorithm. Finally, initial results are presented before pointers to future work and conclusions are given.

9.1 Related Work

In this section related work within the field of automated design space exploration will be briefly discussed. The field of multi-objective optimization using evolutionary algorithms has received a lot of attention within the last decades; however, the focus here will be on applications within the field of automated design space exploration of embedded systems. An excellent introduction to the

field of multi-objective optimization using evolutionary algorithms is given in e.g. [54], other prominent surveys include [21, 105, 106].

Platune [37] requires the designer to specify parameter inter-dependencies and then automatically analyses the permutations of these parameters in order to find the best possible solution. Parameters are inter-dependent if changing the value of one parameter impacts the optimal value of the other. The authors state that such inter-dependent parameters are found rarely in SoC platforms and thus utilize this to find an approximated Pareto set of solutions by exhaustively searching sub-blocks of the design space based on the ordering of the parameter inter-dependency information. The current version of Platune is bound to a MIPS based platform and does not allow other system models to be used.

In [22], Sesame, a performance estimation framework which includes automated design space exploration for multimedia systems, i.e. applications which are stream based, is outlined. The approach to automated design space exploration taken is very similar to the one taken in this thesis and the authors also use evolutionary algorithms for analyzing the design space. In particular, the focus is on solving the mapping problem, i.e. mapping the tasks of an application onto the processing elements of the target architecture.

In [83], a modular design space exploration framework is presented in which the system model, exploration algorithms and objectives are specified individually. Three different heuristics for multi-objective optimization are provided and the flexible structure is very similar to the approach taken in this thesis. However, only limited information is available regarding the actual system model specification and simulation in order to associate cost metrics with the given solution.

CHARMED is presented in [49], which is a framework for multi-objective design space exploration. Applications are represented by task graphs. As part of the search process, the tasks of the task graphs are then automatically mapped to the processing elements and communication resources available as specified by the designer in the architecture model. For fitness evaluation, analytical cost estimates are computed and used to evaluate the given solution. The result of the search is a set of non-dominated solutions which describe the best found trade-offs between the targeted design objectives.

The design space exploration framework presented here, by itself, is not unique but the combination of the automated design space exploration framework with the presented system level modelling and performance estimation framework provides a strong tool for exploring complex designs in a semi-automated fashion with no restrictions on the type of applications which can be considered as is seen in several of the previous efforts within this area.

9.2 Multi-objective optimization

This section will introduce the basic concepts of multi-objective optimization relevant to the design space exploration problem considered here. The definitions in this section are based on the definitions given in [25].

A major difference between a multi-objective optimization problem and a single-objective optimization problem is that the objective space is multi-dimensional. Thus, the objective function maps a decision vector $\vec{x} = (x_1, x_2, ..., x_N)^T$ from the decision variable space X into an objective vector $\vec{z} = (z_1, z_2, ..., z_M)^T$ in the objective space Z. Where N denotes the number of decision variables and M denotes the number of objective functions. Please note that in the following it is assumed that objective functions are to be minimized without a loss of generality.

Definition 9.1 In general, a multi-objective optimization problem with N decision variables and M objective functions is defined as:

$$\begin{aligned} &\text{Minimize} && f(\vec{x}) = (f_1(\vec{x}), f_2(\vec{x}), ..., f_M(\vec{x})) \\ &\text{subject to} && \vec{x} \in X \end{aligned}$$

■

In the following, a particular mapping of a decision variable vector from the decision variable space into an objective vector from the objective space is referred to as a *solution*.

Domination

A key-concept, used in multi-objective optimization for comparing different solutions and assessing their quality, is the concept of domination. In our case, all objective functions are to be minimized, giving us the following definition of domination:

Definition 9.2 A solution x_i dominates x_j ($x_i \preceq x_j$) if both of the following two conditions are satisfied:

1. The solution x_i is at least as good as x_j in all objectives:
   $f_k(x_i) \leq f_k(x_j)$ for all $k = 1, 2, ..., M$
2. The solution x_i is strictly better in at least one objective:
   $f_k(x_i) > f_k(x_i)$ for all $k = 1, 2, ..., M$

■

Where M is the number of objective functions. The dominance definition thus provides a measure of which of the two solutions is the better one.

Non-dominated set

In general, as already mentioned in the introduction of this chapter, no single optimal solution exists when all objectives are equally important; instead a set of non-dominated solutions can be found which each represent a trade-off between the objective functions. This set is referred to as a *non-dominated set* or a *Pareto set*.

Definition 9.3 In a given set of solutions, P, the Pareto set P^* is the set of solutions which are not dominated by any other solution in P. ■

If the entire objective space is searched exhaustively, the resulting Pareto set is said to be the *global Pareto-optimal set*. In practice, however, the global Pareto-optimal set is seldom identifiable due to the vast size of the decision and objective space which is encountered in real world problems. In the following both the terms Pareto set and Pareto front will be used.

9.3 The design space exploration framework

The current version of the automated design space exploration framework is implemented in Java [73]. The framework consists of a core which allows the specification of an optimization problem which is to be solved, the multi-objective optimization algorithm to use, the decision variables and a range of valid values for each. Having specified these, the core is capable of automatically generating system configurations which are used for instantiating a system model for fitness evaluation of the individually generated solutions using the system level modelling and performance estimation framework presented in this thesis.

The current version of the framework assumes the use of multi-objective optimization algorithms which can be categorized as evolutionary algorithms. Currently, only the NSGA-II [26] multi-objective optimization algorithm is implemented but the framework allows other algorithms to be used easily requiring only that a specific Java interface is implemented in order to be used in the framework. Similarly, extensions of the default set of evolutionary operators

for mutation, cross-over, selection etc. can be added, under the constraint that appropriate operator interfaces are implemented.

The current version of the design space exploration framework assumes that all required system model components are already available. Thus no models are generated automatically; instead, only parameters of the individual models are tuned and different mappings are explored. The designer must specify an application model which captures the application under investigation as one or more parallelly executable tasks including their possible data dependencies.

A least one platform model must also be specified by the designer. The platform model specifies the service models of which it is composed as well as the way they are inter-connected. It is possible to specify multiple platform models to be used if several structurally different platforms are to be explored. The design space exploration framework does not automatically create platform models but instead selects one of the possible platform models and then configures it. This includes a selection of individual service models of the platform model if alternatives are allowed. All this information is represented as decision variables in the framework. Each decision variable can be specified in such a way that the possible values can be constrained.

It is also assumed that appropriate cost models are supplied by the designer, so that the objective functions specified can be evaluated correctly. In this way a simulation of a system model configured by the design space exploration framework is evaluated, using the system level modelling and performance estimation framework, allowing the fitness of a solution generated by the design space exploration framework to be evaluated.

9.4 Elitist Non-dominated Sorting Genetic Algorithm

In the current version of the design space exploration framework, a multi-objective evolutionary optimization heuristic, called *Elitist Non-dominated Sorting GA (NSGA-II)* [26], was implemented and used to solve the multi-objective optimization problem faced.

The NSGA-II is a evolutionary algorithm which finds multiple Pareto sets. The heuristic uses an elitist approach by continually emphasizing the Pareto sets, combined with diversity preserving methods, by using a tournament selection operator based on the crowding distance metric [93]. The crowding distance is a measure of how close other solutions are to a specific solution. Using the crowding distance metric, the crowded tournament selection operator is defined as:

Definition 9.4 The crowded tournament selection operator is defined as: Solution s_i wins a tournament with solution s_j if either one of the following conditions is satisfied:

1. Solution s_i has better rank than solution s_j: $r_i > r_j$
2. Solution s_i has the same rank as solution s_j, but better crowding distance: $r_i = r_j$ and $d_i > d_j$

■

The goal of using the crowding distance metric is to obtain a good spread of solutions along the currently best Pareto front found.

The pseudo-code for the NGSA-II algorithm implemented is given in algorithm 1. The NSGA-II heuristics start by forming the off-spring population Q_t from the parent population P_t using crowded tournament selection, crossover and mutation operators [93]. The union of the parent population P_t and the off-spring population Q_t is then formed and the set of non-dominated solutions of the resulting set of size $2N$ is identified. The next step is to cut down the size of the new set to only N solutions. This is done by selecting the N solutions from the best non-dominated front identified in the set formed by the union of the parent and offspring solutions, followed by the second non-dominated front and so on until N solutions have been selected, forming the parent set of the next generation P_{t+1}. The remaining fronts are then simply discarded. However, in most cases when selecting the last solutions, the number of solutions in the last non-dominated front considered is larger than the number of solutions needed to reach N. In this case, a special diversity preserving mechanism is used so that the solutions, which are chosen to be part of P_{t+1}, are selected based on the crowding distance in order to maximize the spread of solutions. The resulting set P_{t+1} now consists of the best solutions of the previous generation parent and offspring. The next generation offspring set Q_{t+1} is then created using the crowded tournament selection, crossover and mutation operators.

The implementation of the NSGA-II algorithm uses the *Non-Dominated-Sorting* algorithm presented in [25]. The pseudo-code for the algorithm implemented is given in algorithm 2. The algorithm starts by finding the first non-dominated front in lines $1 - 16$, and then continues to identify higher level fronts in the while loop in lines $17 - 29$.

Representation of decision variables

The decision variables can be encoded arbitrarily using Java types, however in most case they are encoded as integers which are then decoded individually for

Algorithm 1 NGSA-II(N, G)

```
1:  P_t := Create-Initial-Polulation(N)
2:  Q_t := Create-OffSpring-Polulation(P_t)
3:  for all G do
4:     R_t := Q_t ∪ P_t
5:     F := Non-Dominated-Sorting(R_t)
6:     P_{t+1} := ∅
7:     while |P_{t+1}| + |F_i| < N do
8:        P_{t+1} := P_{t+1} + F_i
9:        i := i + 1
10:    end while
11:    if |P_{t+1}| < N then
12:       Crowding-Sort(F_i)
13:       |P_{t+1}| := |P_{t+1}|∪ (N − |P_{t+1}|) first of F_i
14:    end if
15:    Q_{t+1} := Create-OffSspring-Polulation(P_{t+1})
16: end for
```

each decision variable specified. The decoding is implementation specific and is specified by the designer. Figure 9.2 illustrates the principle. In this case, the application "task mappings" specifies an id of the processing element onto which they are mapped; constraints can be applied in order to determine the processing elements that are valid for mapping. PE specifies the number of processing elements in the platform and the type of each processing element is specified next. For each processing element there exists a set of types which are valid for that particular processing element as illustrated in the figure. Related genes are grouped together in order to preserve locality during the application of the genetic operators when forming offspring populations.

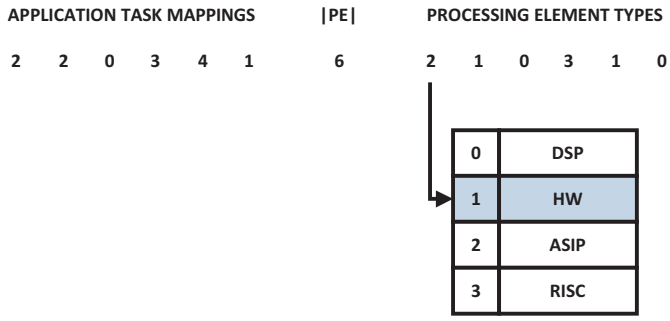


Figure 9.2: Example of the coding of the decision variables of a specific problem.

Algorithm 2 Non-Dominated-Sorting(P)

```
1: for all i ∈ P do
2:    S_i := ∅
3:    n_i := 0
4:    for all j ∈ P do
5:       if j ≠ i then
6:          if i ⪯ j then
7:             S_i = S_i ∪ {j}
8:          else if j ⪯ i then
9:             n_i := n_i + 1
10:         end if
11:      end if
12:   end for
13:   if n_i = 0 then
14:      F_1 := F_1 ∪ {i}
15:   end if
16: end for
17: k := 1
18: while F_k ≠ ∅ do
19:   F_{k+1} := ∅
20:   for all i ∈ F_k do
21:      for all j ∈ S_i do
22:         n_j := n_j − 1
23:         if n_j = 0 then
24:            F_{k+1} := F_{k+1} ∪ {j}
25:         end if
26:         k := k + 1
27:      end for
28:   end for
29: end while
```

Selection

Currently, only the crowded tournament selection operator, defined in definition 9.4 is used for selection of candidate solutions for generating the offspring population.

Crossover

The first version of the framework uses a simple single point crossover operator in which the crossover point is chosen randomly. Only the probability of a crossover needs to be specified as a parameter in this case.

Mutation

A bit-flip-like mutation operator is used in the initial version of the design space exploration framework. The value of the decision variables are changed randomly within their valid range. This leaves only one parameter to be tuned: The mutation probability.

Repair

In order to ensure that only valid permutations of the decision variables are sent for evaluation, a simple repair principle is used which corrects the values of illegal decision variable values through saturation to the upper or lower bound of the decision variable.

Evaluation

The evaluation of each generated solution is by far the most costly operation in the targeted application of the framework for solving the multi-objective optimization problem. Each solution formed, using the evolutionary operators, specifies a system configuration of a system model of the embedded system which is under investigation. The configuration is used as input to the simulation based system level modelling and performance estimation framework. The time used for these simulations range from seconds to several hours. It is therefore obviously preferable that the number of different solutions which needs to be investigated can be minimized.

9.5 Experimental Results

This section is merely an appetizer, illustrating the potential usefulness of having available an automated design space exploration framework as the one proposed in this chapter.

In this problem, the application presented in chapter 7, is considered again. In order to limit the problem only one of the two stereo audio channels is considered. This implies that the application model is composed of six tasks only, in which four tasks are modelling the actual application, i.e. the processing of the stereo audio channel, and two tasks are used for modelling the environment representing a sourcing and sinking of audio samples. The system experiences a real-time constraint, as explained in chapter 7, limiting the total allowed time used to process each sample. This constraint is determined by the sample rate of the incoming audio channel which requires that for every $T_{sample} = 1/F_{sample}$, two processed audio samples must be ready per stereo channel. As explained in chapter 7, the clock frequency of the system components as well as the buffering of audio samples are two ways of tuning the amount of processing which can be applied to the audio channel in order to fulfil the real time constraint. In the current problem, we will only consider a tuning of the clock frequency of the platform components in order to ensure that enough time for the required processing is available. A higher clock frequency implies that more cycles are available for processing but also has higher power consumption as a consequence.

The problem considered has two objectives:

1. Maximize the utilization of the selected platform.
2. Minimize the power consumption.

In this respect, we define the utilization of a platform as the average utilization of each processing element and require that the total utilization of a platform is less than 80% and that no single processing element has a utilization ratio which is higher than 80%. A very rough power model is used to quantify the power consumption based on a pre-characterization of the individual tasks on each processing element and then scaled according to the specific clock frequency. As a rough estimate, it is assumed that power is only dissipated while the processing element is actively performing computations, i.e. clock gating is applied so that the processing element is not clocked when idle. The current implementation of the design space exploration framework assumes that all objective functions are to be minimized; hence the first objective of maximizing the utilization of the investigated platforms is re-written so that we will strive to minimize the total idle time of the platform in order to obtain an objective function which needs to be minimized.

As was the case in the case-study performed in chapter 7, a platform consisting

of no more than four processing elements is considered. Again, three different processing element types are available: A DSP, the application specific ASIP referred to as the SVF processor and a dedicated hardware implementation. The clock frequency of each processor can be controlled individually and five different levels are supported.

Thus, the problem considered here considers both the mapping problem of determining on which processing elements each of the four tasks should be executed as well as how the best platform should be composed in terms of the number of processing elements as well as processing element types and the clock frequency of each. The problem only considers the time required for computation and not communication, because direct point-to-point FIFO-buffers are used between each task in the implementations. In future case-studies, of course, this could be included, too.

In order to assess the quality of the Pareto set found using the design space exploration framework, several approaches can be taken in order to quantify the results in order to perform a comparison of different multi-objective optimization algorithms as well as for the purpose of tuning the operators used, e.g. as proposed in [25]. Metrics such as accuracy expressed as the difference from the optimal Pareto front, distribution of solutions along the best Pareto front found as well as the extent of the solution, i.e. the values covered, are suggested for use. However, in most cases the optimal Pareto front is not known and instead the best found front of all runs is most often used as reference when measuring the quality of the individual runs which have different parameters and/or uses a different algorithm.

However, in the problem considered here, the design space is searched exhaustively in order to obtain a good reference. This can be done because of the relative small size of the problem considered in which a total of 51,840,000 different solutions exist based on a direct permutation of the decision variables as illustrated in table 9.1. However, a substantial amount of these solutions can be disregarded due to invalid combinations of decision variables. As an example, the tasks of the application model are only allowed to be mapped to physical processing elements of the current platform instance. In this case, this first pruning of the design space was performed analytically so that invalid permutations were not used for simulation. Still 13,236,990 solutions were valid and required to be simulated in order to search the full design space. A number of these simulations violated the real time requirement, implying that not enough time was available for the required processing to finish. Also, several of the valid permutations of decision variables map to the same point in the objective space, limiting the number of valid and unique solutions to only 26,315. Even though, the number of valid and unique solutions seems rather small, in order to identify these, the simulation of 13,236,990 solutions was still required in order to search the entire design space of this small problem exhaustively.

Permutations	|PE|	T0	T1	T2	T3	PE0	PE1	PE2	PE3	CLK0	CLK1	CLK2	CLK3	Total
Values	4	4	4	4	4	3	3	3	3	5	5	5	5	51,840,000
Valid:														
$\|PE\| = 1$		1	1	1	1	3				5				15
$\|PE\| = 2$		2	2	2	2	3	3			5	5			3,600
$\|PE\| = 3$		3	3	3	3	3	3	3		5	5	5		273,375
$\|PE\| = 4$		4	4	4	4	3	3	3	3	5	5	5	5	12,960,000
Total														13,236,990

Table 9.1: Possible values of decision variables. |PE| expresses the number of processing elements of the platform. T0-T3 determines the possible mapping of the specific task to a processing element. PE0-PE3 determines the type of processing element and finally CLK0-CLK3 determines the clock frequency of the processing element. The first row expresses all permutations of the decision variables. In the next four rows, the valid number of permutations is identified for each platform consisting of 1 to 4 processing elements respectively.

All simulations were carried out using the latency based service models of the processing elements which were presented in chapter 7 and it should be noted that all simulations were performed on an Intel Core 2 Duo processor running 2.0 GHz and with 2GB RAM. The utilization is measured as the time each processing element of a platform is active. Each simulation had a duration of 100 ms of simulation time in which a total of 4800 audio samples were processed with a sample rate $F_{sample} = 48kHz$. The total time required to perform the simulations required to do an exhaustive search of the design space, using the system level modelling and performance estimation framework presented in part I and II, was roughly 37 hours! It is obvious that the time required for performing a full search of the design space is not feasible already when considering problems which have only a few more decision variable values. This is where the multi-objective optimization algorithms can play an interesting role as indicated by the results presented in the following.

Figure 9.3 shows a plot of all evaluated valid solutions in terms of the estimated power consumption vs. the inverse utilization of the specific platform (i.e. the idle time) with each solution indicated by a gray cross. A value of 20% on the inverse utilization axis thus corresponds to a utilization ratio of 80%. The figure also shows the found Pareto front using the NSGA-II algorithm. Qualitatively, it can be seen from the figure that the Pareto front found using the NSGA-II algorithm is, in fact, covering solutions which represent optimal trade-offs between the two objectives.

It is very interesting to see that, in this case, only 100 generations, each with a population size of 30 individuals is required in order to obtain this Pareto front. This implies that only 3000 solutions have been evaluated. Furthermore, the time required to perform this search was approximately 10 minutes! Naturally, there is a trade-off between the number of solutions investigated and the quality of the best found Pareto front. The number of solutions investigated is determined by the product of the population size and the number of generations of which the algorithm is allowed to run and as these increase, a larger part of the design space can be explored implying that the probability of finding better solutions increases at the expense of the time required to perform this search.

In general, the results show that with the current partitioning of tasks and the available processing elements, it is hard to utilize the investigated platforms efficiently as most solutions lie in the range of 10%-60% utilization. However, there are some very interesting solutions which lie in the lower left half of figure 9.3, and this is also were the Pareto front found using the design space exploration framework is located. These solutions have a high utilization ratio while still having low power consumption. Looking into the system instances behind these solutions, it was seen that most of these solutions are based on platforms consisting of either two ASIPs, the DSP or a direct hardware imple-

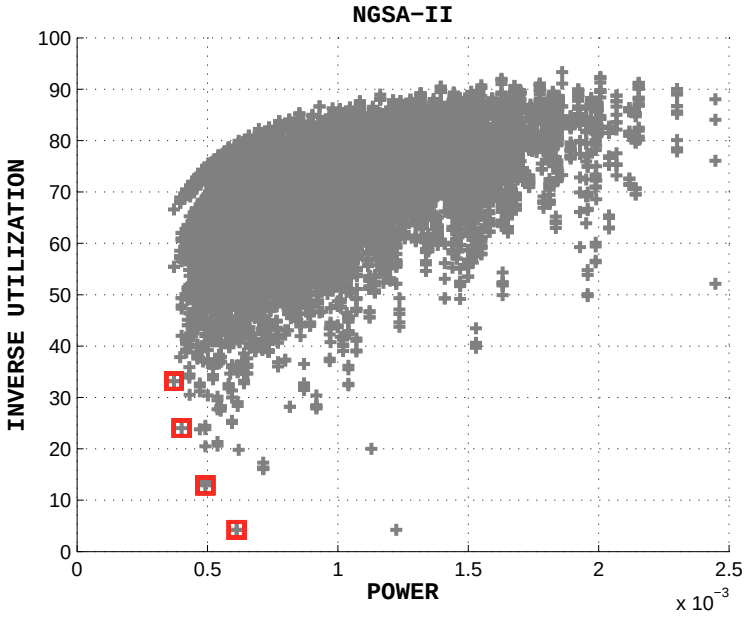


Figure 9.3: Population size=30, Generations = 100, Mutation probability = 0.1, Crossover probability = 0.1. The mutation and crossover operators used are the ones described in section 9.4.

mentation running at different clock frequencies. In particular, the Pareto front is composed of platforms composed of two ASIP processing elements running either both 12.5 MHz or one at 25MHz and the other at 6.25 MHz depending on the mapping of tasks. Also, the Pareto front suggests that a platform consisting of a single DSP processor running at 12.5 MHz is also attractive with the given objectives. The hardwired platform is not part of the Pareto front with the given objectives; if the cost of silicon area or component cost had been part of the objectives, however, this outcome had probably been different. The great thing about programmable platforms, on the other hand, is the flexibility of the implemented silicon solution in which changes and bug fixes are much easier to accommodate compared to a hardwired platform and this puts extra weight on the benefits of choosing one of the platforms found in this design space exploration example.

It should be noted that the performance of the design space exploration framework is very much dependent on the chosen parameters and operators for the selected evolutionary algorithm. With the current investigations and problems examined, no general guidelines can be presented but it is the hope that a default parameter set could be provided in the future. However, one major difficulty is

that, in general, the optimal Pareto set is not known and if the quality of a given configuration of the algorithm used is how fast it can find the near-to-optimal Pareto set, one does not know how close the approximated Pareto front is to the optimal one unless the entire design space is explored. Instead, different parameter sets can, of course, be compared relatively so that they are allowed to run for the same amount of time, or for the same number of generations, and then the one with the best found Pareto set is preferred. However, as mentioned already, it may very likely be the case that this is also highly related to the specific problem considered.

9.6 Summary

This chapter has introduced the initial investigations of the coupling of the system level modelling and performance estimation framework presented in this thesis with an extension which allows automated design space exploration to be carried out.

The current version of the automated design space exploration framework uses the evolutionary multi-objective optimization algorithm NSGA-II [26] for automated design space exploration. However, the design space exploration framework is constructed in such a way that different meta-heuristics can easily be implemented and used instead, if preferred. Similarly other types of selection operators and types of genetic operators can be implemented if needed.

In the category of future work, it would be interesting to investigate the possibility of representing the system models used for performance estimation, modelled at two levels of abstraction, so that a high level and a low level version would exist, allowing rough estimates to be produced fast using the high level model, and detailed estimates to be produced using the low abstraction level version. In this way the potential candidate solutions which were to be selected could be investigated using the high level version in order to speed up the search process and, then, use the detailed version only for those solutions which were selected as the best candidates of the current population and which are used to form the next generation of solutions.

Finally, it was shown how the design space exploration framework was applied to a case-study from Bang & Olufsen ICEpower. It is obvious from the presented experimental results that much work is still required in order to verify the benefits of the approach in more detail which require that larger and more complex case-studies should be performed in order to allow a further assessment of the framework. However, the initial results obtained from the relative small example, which was presented, are encouraging.

Part IV

Perspectives and Conclusions

Chapter 10

Conclusions and Outlook

This thesis presents the work that has been carried out during the course of the Ph.D. project with the title *System Level Modelling and Performance Estimation of Embedded Systems*. Efficient system level modelling and performance estimation of embedded systems is required in order to tackle the increasing design complexity associated with embedded systems of today to allow true design space exploration to be carried out.

The main contribution of the thesis is the presented framework for system level modelling and performance estimation. As discussed, the framework does not strictly enforce the use of a specific design methodology but is related to the Y-chart approach [12, 50] and leverage principles of the Platform Based Design (PBD) paradigm [48]. As a consequence, a separate specification of the application (functionality) and the target architecture (implementation) is used in the framework in the form of an application model and platform model respectively. Consequently, the design methodologies introduced are based on these principles. The framework is capable of capturing embedded systems at the system level, allowing designers to associate quantitative performance estimates obtained through simulations with the individual system components, in this way allowing designers to select the best suited system from a well-defined criteria. The framework allows a given embedded system to be constructed and explored before a physical realization is present.

Fundamental to the framework is the concept of service models. The service model can be seen as a meta-model which allows a unified modelling of both hardware and software components. Services are used to represent the functionality offered by a component and cost, and the implementation of the function-

ality modelled, can be associated with the service through an implementation of the service. Gradual refinement of models is supported providing a very flexible framework. In principle, the service model concept allows arbitrary models-of-computation to be used for capturing the behaviour of the individual components and also allows communication across abstraction levels using the notion of service requests and, so, makes it possible to have components described at multiple levels of abstraction to co-exist within the same model instance. In the practical realization of the current framework, restrictions exist on the types of models of computation which can be implemented. These are inferred due to the chosen discrete event simulation engine used, and the representation of time, implying that new models of computation can be added under the constraint that they must be implementable in this context and implement inter-model communication through service requests.

One example of a model-of-computation for synchronous hardware modelling fitting into the service model framework was also presented which has showed promising results with respect to handling model complexity, user friendliness for capturing synchronous hardware descriptions and increasing the obtainable simulation speed compared to register transfer level simulations, while still being cycle accurate and bit true as was illustrated in the industrial case-study presented in chapter 7.

In addition to this, an initial version of a high level description language for automatically generating simulation models using the presented model-of-computation was also introduced, easing the practical specification of models which is of great importance in the context of industrial use at e.g. Bang & Olufsen ICEpower. Finally, an initial version of an automated design space exploration framework which used the presented system level modelling and performance estimation framework was outlined. The design space exploration framework is based on the use of evolutionary multi-objective algorithms and is constructed in such a way that new evolutionary operators and algorithms can be easily implemented. Currently, only one specific algorithm is implemented and a small case-study was performed in order to illustrate the usefulness of such a framework. Much work still lies ahead in order to verify initial results on large scale problems.

The primary benefits of the framework and components described above are the possibilities of exploring a large number of candidate systems within a short time frame leading to better designs, easier design verification through a possible iterative refinement of the executable system description, and finally the possibility of a reduction of the time-to-market of the design and implementation of the system under consideration.

10.1 Limitations of the approach

The work presented in this thesis has showed great potential. However, it is by no means the solution to all problems experienced within the field of embedded systems design and thus several challenges still need to be addressed.

Two categories of challenges exist for the system level modelling and performance estimation framework presented in this thesis. The first relates to the fundamental concepts and the second is of a more practical nature.

The focus has been on outlining a framework for practical applicability in the industry and, so, a somewhat pragmatic approach has been taken. Thus, currently, no guarantees can be given regarding the general applicability of the framework and the realization of a more formal foundation would be interesting the future.

In the practical category, a major limitation of the currently realized version of the presented framework is the proprietary specification of models. All models must be described in Java using the current realization of the framework - and no import possibilities currently exist for automatic conversion of models described in other languages. Thus, additional time spent on software development in order to provide commercial- quality tools supporting the method will be required. Nevertheless, the benefits of having such a framework available will be twofold. Firstly, improved productivity will be possible, and secondly, it will be possible to obtain better designs because a larger part of the design space will be explored. Verification of the individual refinement steps will also be easier and, last but not least, changes to the design will be easier to accommodate.

The general service based concept scales very well due to the inherent abstraction level refinement and hierarchical support. The practical realization of the concepts in the framework presented, however, is expected to suffer from scalability problems. Model construction in itself can easily accommodate complex and large models; however, the current simulation engine is expected not to scale very well, among other things due to the single threaded implementation of the simulation kernel. This is due to the discrete event simulation engine which uses a global notion of time. Several approaches have been explored in the past in order to parallelize discrete event simulation engines allowing time to be synchronized, e.g. only on communication between models. Such approaches could be interesting to investigate in the future in order to obtain a more efficient simulation engine. Another approach would be to adopt an existing discrete event simulation engine such as the one provided with e.g. SystemC. In practice, this would be possible simply by providing a SystemC library which would provide the general service model concepts for a practical construction of models. Such an approach would allow designers to specify models quite similar to their accustomed SystemC models, only now using the service request based approach

for inter-model communication as well as the interface based approach which is the fundamental element behind the abstraction level refinement capabilities defined by the framework.

10.2 Future Work

As already discussed in the previous section, there are, of course, a number of limitations to the framework presented.

In the category of practical related future work the focus is on the usage of the framework - it is very important that sufficient quality tools are available in order for a framework, as the one presented, to be used in practice. In general, automation of the design steps is not a goal in itself; however, it is very important that users of a framework, and the tools offered with this, initially are able to control all steps in detail. In later stages, automation of the time consuming cumbersome steps is of course highly desirable. Automatic correct-by-construction code generation for both hardware and software models would be interesting to investigate. In principle, communication refinement could also be performed automatically utilizing the information of the service models which specifies inter-model communication through an interface based approach combined with the use of service requests for modelling communication transactions. The model based design approach taken by the framework should also make it possible to allow a systematic generation of test benches and test vectors for input, allowing a better verification of systems. Several analysis tools could be developed as well - e.g. tools for automatic identification of mappable components based on the set of offered services.

The presented system level modelling and performance estimation framework provides an infrastructure for designers to use when modelling systems and estimating the performance of these. The thesis has not focused on developing actual cost models, e.g. power models etc., for performance estimation. These, of course, play a vital role in system design and, thus, a great amount of work could lie in the category of developing or adapting cost models for performance estimation.

Finally, in the category of future work, it is obvious that in order to asses the ideas presented here, more elaborate case-studies are required in order to verify the usefulness of the framework in order to identify further limitations than the ones already outlined.

Part V

Appendix

Appendix A

Producer-Consumer Example Source Code

```
class Producer extends AbstractServiceModel {

    private IServiceModelInterface fWrite;

    private final IntValue fValue = new IntValue(0);

    private IServiceRequest fSR;

    public Producer(ISimulator simulator, IServiceModelInterface write) {
      super("PRODUCER");

      this.setSimulator(simulator);

      fWrite = write;

      fSR = fWrite.createServiceRequest("WRITE", new Object[] { fValue });
    }

    /*
     * (non-Javadoc)
     *
     * @see com.sysmopee.core.model.process.AbstractProcess#init()
     */
    public final void init() {
      this.fNextBlock = fBlock0;

      this.setActive();
    }
```

```
  private final IBlock fBlock0 = new IBlock() {
    /*
     * (non-Javadoc)
     *
     * @see com.sysmopee.core.model.process.IBlock#execute()
     */
    @Override
    public final IBlock execute() {
      if (DEBUG) {
        System.out.println("Producer starting @ "
            + fSimulator.getSimulationTime());
      }

      // Request blocking write
      fWrite.request(fSR);

      waitFor(fSR, IServiceRequest.EventType.DONE);

      return fBlock1;
    }
  };

  private final IBlock fBlock1 = new IBlock() {
    @Override
    public final IBlock execute() {
      if (DEBUG) {
        System.out.println("Producer done @ "
            + fSimulator.getSimulationTime());
      }

      waitFor(0);

      return fBlock0;
    }
  };
}

class Consumer extends AbstractServiceModel {

  private IServiceModelInterface fRead;

  private final IntValue fValue = new IntValue(21);

  private IServiceRequest fSR;

  public Consumer(ISimulator simulator, IServiceModelInterface read) {
    super("CONSUMER");

    this.setSimulator(simulator);

    fRead = read;

    fSR = fRead.createServiceRequest("READ", new Object[] { fValue });
  }
```

```
  public void init() {
    super.init();

    this.fNextBlock = fBlock0;

    this.setActive();
  }

  private final IBlock fBlock0 = new IBlock() {
    @Override
    public final IBlock execute() {
      if (DEBUG) {
        System.out.println("Consumer starting @ "
            + fSimulator.getSimulationTime());
      }

      // Request blocking read
      fRead.request(fSR);

      waitFor(fSR, IServiceRequest.EventType.DONE);

      return fBlock1;
    }
  };

  private final IBlock fBlock1 = new IBlock() {
    @Override
    public final IBlock execute() {
      // DO SOME STUFF WITH THE DATA
      int val = fValue.getValue();

      if (DEBUG) {
        System.out.println("Consumer received: " + val + " @ "
            + fSimulator.getSimulationTime());
      }

      waitFor(0);

      return fBlock0;
    }
  };
}
```

Appendix B

SVF Processor instruction format

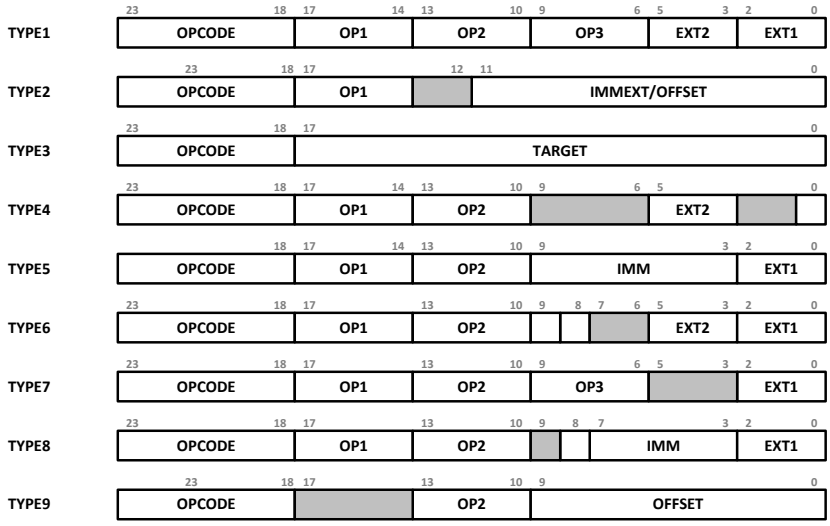


Figure B.1: Overview of the SVF instruction format.

Appendix C

SMDL description of the SVF processor

This appendix holds the SMDL source code of the SVF application specific processor.

```
servicemodel SVF{
  /* The structure of the SVF Core */
  structure{
    /* Active interfaces */
    ActiveInterface SOURCE_LEFT;
    ActiveInterface SOURCE_RIGHT;

    ActiveInterface SINK_LEFT;
    ActiveInterface SINK_RIGHT;

    ActiveInterface I2C;

    word<24> mem[4096];

    /* Register file */
    word<24> reg[16];

    /* Filter Register file */
    word<24> a_lp[16];
    word<24> a_bp[16];
    word<24> a_hp[16];

    word<24> b_lp[16];
    word<24> b_bp[16];
```

```
    word<24> b_hp[16];

    word<24> c1[16];
    word<24> c2[16];
    word<24> fcfg[16];

    /* Configuration and status registers */
    word<24> cfgst[8];

    word<16> pc;
  }

  /* Encoding of the SVF Instruction Set */
  encoding{
    OPCODE[23:18];

    OP1[17:14];
    OP2[13:10];
    OP3[9:6];

    IMMEEXT[11:0];
    IMMEXT[9:3];
    IMM[7:3];

    TARGET[17:0];

    EXT1[2:0];
    EXT2[5:3];

    OFFSET[9:0];

    EEXT1[8];
    EEXT2[9];
  }

  abstract service EXTENDEDTYPE1{

    decode word<24> getOP1(){
      switch(EXT1){
        case 0:
          return reg[OP1];
        break;

        case 1:
          return cfgst[OP1];
        break;

        case 2:
          return a_lp[OP1];
        break;

        case 3:
          return a_bp[OP1];
        break;
```

```
        case 4:
          return a_hp[OP1];
        break;

        case 5:
          return b_lp[OP1];
        break;

        case 6:
          return b_bp[OP1];
        break;

        case 7:
          return b_hp[OP1];
        break;
      }
    }
  }

  abstract service EXTENDEDTYPE2{

    decode word<24> getOP1(){
      switch(EXT1){
        case 0:
          return reg[OP1];
        break;

        case 1:
          return cfgst[OP1];
        break;

        case 2:
          return a_lp[OP1];
        break;

        case 3:
          return a_bp[OP1];
        break;

        case 4:
          return a_hp[OP1];
        break;

        case 5:
          return b_lp[OP1];
        break;

        case 6:
          return b_bp[OP1];
        break;

        case 7:
          return b_hp[OP1];
        break;
```

```
    }
  }

  decode word<24> getOP2(){
    switch(EXT2){
      case 0:
        return reg[OP2];
      break;

      case 1:
        return cfgst[OP2];
      break;

      case 2:
        return a_lp[OP2];
      break;

      case 3:
        return a_bp[OP2];
      break;

      case 4:
        return a_hp[OP2];
      break;

      case 5:
        return b_lp[OP2];
      break;

      case 6:
        return b_bp[OP2];
      break;

      case 7:
        return b_hp[OP2];
      break;
    }
  }
}

abstract service EEXTENDEDTYPE1 extends EXTENDEDTYPE1{

  decode word<24> getEOP1(){
    if(EEXT1 == 1){
      switch(EXT1){
        case 0:
          return c1[OP1];
        break;

        case 1:
          return c2[OP1];
        break;

        default:
          return fcfg[OP1];
```

```
        break;
      }
    }
    else{
      return getOP1();
    }
  }

}

abstract service EEXTENDEDTYPE2 extends EXTENDEDTYPE2{

  decode word<24> getEOP1(){
    if(EEXT1 == 1){
      switch(EXT1){
        case 0:
          return c1[OP1];
        break;

        case 1:
          return c2[OP1];
        break;

        default:
          return fcfg[OP1];
        break;
      }
    }
    else{
      return getOP1();
    }
  }

  decode word<24> getEOP2(){
    if(EEXT2 == 1){
      switch(EXT2){
        case 0:
          return c1[OP1];
        break;

        case 1:
          return c2[OP1];
        break;

        default:
          return fcfg[OP1];
        break;
      }
    }
    else{
      return getOP2();
    }
  }
}
```

```
  service NOP{
    encoding{
      OPCODE = 0, 18'b0;
    }

    /* BEHAVIOUR */
    void p1(){
      // DO NOTHING
    }
  }

  /*****************************************
  * START OF TYPE1 INSTRUCTIONS
  *****************************************/
  abstract service TYPE1 extends EXTENDEDTYPE2{
    encoding{
      OPCODE, OP1, OP2, OP3, EXT2, EXT1;
    }

    /* BEHAVIOUR */
    abstract void p1();
  }

  service ADDS extends TYPE1{
    encoding{
      OPCODE = 1;
    }
    void p1(){
      getOP1() = getOP2() + reg[OP3];
    }
  }

  service ADDU extends TYPE1{
    encoding{
      OPCODE = 2;
    }

    void p1(){
      getOP1() = getOP2() + reg[OP3];
    }
  }

  service MULS extends TYPE1{
    encoding{
      OPCODE = 3;
    }

    void p1(){
      getOP1() = getOP2() * reg[OP3];
    }
  }

  service MULU extends TYPE1{
    encoding{
      OPCODE = 4;
```

```
    }

    void p1(){
      getOP1() = getOP2() + reg[OP3];
    }
  }

  service MACS extends TYPE1{
    encoding{
      OPCODE = 5;
    }

    void p1(){
      getOP1() = getOP2() * reg[OP3] + getOP1();
    }
  }

  service MACU extends TYPE1{
    encoding{
      OPCODE = 6;
    }

    void p1(){
      getOP1() = getOP2() * reg[OP3] + getOP1();
    }
  }

  service AND extends TYPE1{
    encoding{
      OPCODE = 35;
    }

    void p1(){
      getOP1() = getOP2() & reg[OP3];
    }
  }

  service OR extends TYPE1{
    encoding{
      OPCODE = 36;
    }

    void p1(){
      getOP1() = getOP2() | reg[OP3];
    }
  }

  service XOR extends TYPE1{
    encoding{
      OPCODE = 37;
    }

    void p1(){
      getOP1() = getOP2() ^ reg[OP3];
    }
```

```
  }

  service LW extends TYPE1{
    encoding{
      OPCODE = 24;
    }
    void p1(){
      if(getOP2()+reg[OP3] < x200){
        // Normal write to memory
        getOP1() = mem[getOP2() + reg[OP3]];
      }
      else if(getOP2()+reg[OP3] == x800){
        SOURCE_LEFT->READ(getOP1());
      }
      else if(getOP2()+reg[OP3] == x801){
        SOURCE_RIGHT->READ(getOP1());
      }
      else if(getOP2()+reg[OP3] >= xA00 && getOP2()+reg[OP3] < xA20){
        I2C->READ(getOP1(), (getOP2()+reg[OP3]) & x1F);
      }
    }
  }

  service SW extends TYPE1{
    encoding{
      OPCODE = 23;
    }
    void p1(){
      if(getOP2()+reg[OP3] < x200){
        // Normal write to memory
        mem[getOP2() + reg[OP3]] = getOP1();
      }
      else if(getOP2()+reg[OP3] == x900){
        SINK_LEFT->WRITE(getOP1());
      }
      else if(getOP2()+reg[OP3] == x901){
        SINK_RIGHT->WRITE(getOP1());
      }
      else if(getOP2()+reg[OP3] >= xA00 && getOP2()+reg[OP3] < xA20){
        I2C->WRITE(getOP1(), (getOP2()+reg[OP3]) & x1F);
      }
    }
  }

  /*****************************************
  * START OF TYPE2 INSTRUCTIONS
  *****************************************/
  abstract service TYPE2{
    encoding{
      OPCODE, OP1, 2'b00, IMMEEXT;
    }

    /* BEHAVIOUR */
    abstract void p1();
  }
```

```
service ADDIS extends TYPE2{
  encoding{
    OPCODE = 7;
  }

  void p1(){
    reg[OP1] = reg[OP1] + SIGNED(IMMEEXT);
  }
}

service ADDIU extends TYPE2{
  encoding{
    OPCODE = 8;
  }

  void p1(){
    reg[OP1] = reg[OP1] + UNSIGNED(IMMEEXT);
  }
}

service MULIS extends TYPE2{
  encoding{
    OPCODE = 9;
  }

  void p1(){
    reg[OP1] = reg[OP1] * SIGNED(IMMEEXT);
  }
}

service MULIU extends TYPE2{
  encoding{
    OPCODE = 10;
  }

  void p1(){
    reg[OP1] = reg[OP1] * UNSIGNED(IMMEEXT);
  }
}

service MOVI extends TYPE2{
  encoding{
    OPCODE = 15;
  }

  void p1(){
    reg[OP1] = (reg[OP1][23:12] & xFFF000) | IMMEEXT[11:0];
  }
}

service MOVUI extends TYPE2{
  encoding{
    OPCODE = 16;
  }
```

```
  void p1(){
    reg[OP1] = (IMMEEXT[11:0] sll 12) | reg[OP1][11:0];
  }
}

service LWA extends TYPE2{
  encoding{
    OPCODE = 28;
  }

  void p1(){
    if(IMMEEXT < x200){
      reg[OP1] = mem[IMMEEXT];
    }
    else if(IMMEEXT == x800){
      SOURCE_LEFT->READ(reg[OP1]);
    }
    else if(IMMEEXT == x801){
      SOURCE_RIGHT->READ(reg[OP1]);
    }
    else if(IMMEEXT >= xA00 && IMMEEXT < xA20){
      I2C->READ(reg[OP1], IMMEEXT & x1F);
    }
  }
}

service SWA extends TYPE2{
  encoding{
    OPCODE = 27;
  }

  void p1(){
    if(IMMEEXT < x200){
      mem[IMMEEXT] = reg[OP1];
    }
    else if(IMMEEXT == x900){
      SINK_LEFT->WRITE(reg[OP1]);
    }
    else if(IMMEEXT == x900){
      SINK_RIGHT->WRITE(reg[OP1]);
    }
    else if(IMMEEXT >= xA00 && IMMEEXT < xA20){
      I2C->WRITE(reg[OP1], IMMEEXT & x1F);
    }
  }
}

service ANDI extends TYPE2{
  encoding{
    OPCODE = 38;
  }

  void p1(){
    reg[OP1] = reg[OP1] & IMMEEXT;
```

```
  }
}

service ORI extends TYPE2{
  encoding{
    OPCODE = 39;
  }

  void p1(){
    reg[OP1] = reg[OP1] | IMMEEXT;
  }
}

service XORI extends TYPE2{
  encoding{
    OPCODE = 40;
  }

  void p1(){
    reg[OP1] = reg[OP1] ^ IMMEEXT;
  }
}

service BGTZ extends TYPE2{
  encoding{
    OPCODE = 20;
  }

  void p1(){
    if(reg[OP1] > 0){
      pc = pc + SIGNED(IMMEEXT);
    }
  }
}

service BLEZ extends TYPE2{
  encoding{
    OPCODE = 21;
  }

  void p1(){
    if(reg[OP1] <= 0){
      pc = pc + SIGNED(IMMEEXT);
    }
  }
}

/*****************************************
* START OF TYPE3 INSTRUCTIONS
*****************************************/
abstract service TYPE3{
  encoding{
      OPCODE, TARGET;
  }
```

```
    /* BEHAVIOUR */
    abstract void p1();
  }

  service JMP extends TYPE3{
    encoding{
      OPCODE = 22;
    }

    void p1(){
      pc = TARGET;
    }
  }

  /*****************************************
  * START OF TYPE4 INSTRUCTIONS
  *****************************************/
  abstract service TYPE4 extends EXTENDEDTYPE2{
    encoding{
        OPCODE, OP1, OP2, 4'b0000, EXT2, EXT1;
    }

    /* BEHAVIOUR */
    abstract void p1();
  }

  service SVF1 extends TYPE4{
    encoding{
      OPCODE = 17;
    }

    /* BEHAVIOUR */
    void p1(){
      a_hp[OP2] = (getOP1() - a_lp[OP2]) * c1[OP2] + a_lp[OP2];
    }
  }

  service SVF2A extends TYPE4{
    encoding{
      OPCODE = 18;
    }

    /* BEHAVIOUR */
    void p1(){
      a_hp[OP2] = (getOP1() - a_lp[OP2]) * c1[OP2] + a_lp[OP2];
    }
  }

  service SVF2B extends TYPE4{
    encoding{
      OPCODE = 19;
    }

    /* BEHAVIOUR */
    void p1(){
```

```
      a_hp[OP2] = (getOP1() - a_lp[OP2]) * c1[OP2] + a_lp[OP2];
    }
  }

  /*****************************************
  * START OF TYPE5 INSTRUCTIONS
  *****************************************/
  abstract service TYPE5 extends EXTENDEDTYPE1{
    encoding{
      OPCODE, OP1, OP2, IMMEXT, EXT1;
    }

    /* BEHAVIOUR */
    abstract void p1();
  }

  service SLLI extends TYPE5{
    encoding{
      OPCODE = 11;
    }

    /* BEHAVIOUR */
    void p1(){
      getOP1() = reg[OP2] sll IMMEXT;
    }
  }

  service SRLI extends TYPE5{
    encoding{
      OPCODE = 12;
    }

    /* BEHAVIOUR */
    void p1(){
      getOP1() = reg[OP2] srl IMMEXT;
    }
  }

  service SRAI extends TYPE5{
    encoding{
      OPCODE = 13;
    }

    /* BEHAVIOUR */
    void p1(){
      getOP1() = reg[OP2] sra IMMEXT;
    }
  }

  service BGT extends TYPE5{
    encoding{
      OPCODE = 31;
    }

    /* BEHAVIOUR */
```

```
    void p1(){
      if(getOP1() > reg[OP2]){
        pc = pc + SIGNED(IMMEXT);
      }
    }
  }

  service BLT extends TYPE5{
    encoding{
      OPCODE = 32;
    }

    /* BEHAVIOUR */
    void p1(){
      if(getOP1() <= reg[OP2]){
        pc = pc + SIGNED(IMMEXT);
      }
    }
  }

  service BEQ extends TYPE5{
    encoding{
      OPCODE = 33;
    }

    /* BEHAVIOUR */
    void p1(){
      if(getOP1() == reg[OP2]){
        pc = pc + SIGNED(IMMEXT);
      }
    }
  }

  service BNE extends TYPE5{
    encoding{
      OPCODE = 33;
    }

    /* BEHAVIOUR */
    void p1(){
      if(getOP1() != reg[OP2]){
        pc = pc + SIGNED(IMMEXT);
      }
    }
  }

  /*****************************************
  * START OF TYPE6 INSTRUCTIONS
  *****************************************/
  abstract service TYPE6 extends EEXTENDEDTYPE2{
    encoding{
      OPCODE, OP1, OP2, EEXT1, EEXT2, 2'b00, EXT2, EXT1;
    }

    /* BEHAVIOUR */
```

```
    abstract void p1();
  }

  service MOV extends TYPE6{
    encoding{
        OPCODE = 14;
    }

    /* BEHAVIOUR */
    void p1(){
      getOP1() = getOP2();
    }
  }

  service ABS extends TYPE6{
    encoding{
        OPCODE = 29;
    }

    /* BEHAVIOUR */
    void p1(){
      if(getOP2() < 0){
        getOP1() = ~getOP2() + 1;
      }
    }
  }

  /****************************************
  * START OF TYPE7 INSTRUCTIONS
  ****************************************/
  abstract service TYPE7{
    encoding{
        OPCODE, OP1, OP2, OP3, 3'b000, EXT1;
    }

    /* BEHAVIOUR */
    abstract void p1();
  }

  /****************************************
  * START OF TYPE8 INSTRUCTIONS
  ****************************************/
  abstract service TYPE8 extends EEXTENDEDTYPE1{
    encoding{
        OPCODE, OP1, OP2, 1'b0, EEXT1, IMM, EXT1;
    }

    /* BEHAVIOUR */
    abstract void p1();
  }

  service LWI extends TYPE8{
    encoding{
        OPCODE = 26;
    }
```

```
    void p1(){
      // Local memory access
      if(reg[OP2] + IMM < x200){
        getEOP1() = mem[reg[OP2] + IMM];
      }
      else if(reg[OP2] + IMM == x800){
        SOURCE_LEFT->READ(getEOP1());
      }
      else if(reg[OP2] + IMM == x801){
        SOURCE_RIGHT->READ(getEOP1());
      }
      else if(reg[OP2] + IMM >= xA00 && reg[OP2] + IMM <= xA1F){
        I2C->READ(getEOP1(), (reg[OP2] + IMM) & x1F);
      }
    }
  }

  service SWI extends TYPE8{
    encoding{
        OPCODE = 25;
    }
    void p1(){
      if(reg[OP2] + IMM < x200){
        // Normal write to memory
        mem[reg[OP2] + IMM] = getEOP1();
      }
      else if(reg[OP2] + IMM == x900){
        SINK_LEFT->WRITE(getEOP1());
      }
      else if(reg[OP2] + IMM == x900){
        SINK_RIGHT->WRITE(getEOP1());
      }
      else if(reg[OP2] + IMM >= xA00 && reg[OP2] + IMM <= xA1F){
        I2C->WRITE(getEOP1(), (reg[OP2] + IMM) & x1F);
      }
    }
  }

  /*****************************************
  * START OF TYPE9 INSTRUCTIONS
  *****************************************/
  abstract service TYPE9{
    encoding{
        OPCODE, 4'b0000, OP2, OFFSET;
    }

    /* BEHAVIOUR */
    abstract void p1();
  }
}
```

Appendix D

Generated Java source code for the SVF processor model

This appendix holds the generated Java source code of the HCPN based service model of the SVF application specific processor.

```
package com.sysmopee.test;

import java.math.BigInteger;

import com.sysmopee.core.model.datatypes.DataWord;
import com.sysmopee.core.model.datatypes.IDataWord;
import com.sysmopee.core.model.servicemodel.AbstractServiceModelAdapter;
import com.sysmopee.core.model.servicemodel.AbstractServiceModelInterface;
import com.sysmopee.core.model.servicemodel.AbstractServiceRequest;
import com.sysmopee.core.model.servicemodel.IEvaluate;
import com.sysmopee.core.model.servicemodel.IService;
import com.sysmopee.core.model.servicemodel.IServiceDoneListener;
import com.sysmopee.core.model.servicemodel.IServiceModel;
import com.sysmopee.core.model.servicemodel.IServiceModelInterface;
import com.sysmopee.core.model.servicemodel.IServiceRequest;
import com.sysmopee.core.model.servicemodel.Service;
import com.sysmopee.core.model.servicemodel.state.AbstractServiceModelState;
import com.sysmopee.core.model.servicemodel.state.IMemory;
import com.sysmopee.core.model.servicemodel.state.IRegister;
import com.sysmopee.core.model.servicemodel.state.IRegisterGroup;
import com.sysmopee.core.model.servicemodel.state.StateFactory;
import com.sysmopee.core.simulator.ISimulator;
import com.sysmopee.model.servicemodel.hcpn.core.AbstractHCPNServiceModel;
import com.sysmopee.model.servicemodel.hcpn.core.IHCPNServiceModel;
import com.sysmopee.model.servicemodel.hcpn.core.IServiceRequestPlace;

public class SVF3 extends AbstractHCPNServiceModel {
  public SVF3() {
    this.setState(new SVF3ServiceModelState());
    this.setServiceModelAdapter(new SVF3ServiceModelAdapter(this));
```

```
}

class SVF3ServiceModelAdapter extends AbstractServiceModelAdapter {

  private IServiceRequestPlace p0;
  private IServiceRequestPlace p1;
  private IServiceRequestPlace p2;

  public SVF3ServiceModelAdapter(IServiceModel p0) {
    super(p0);
  }

  /*
   * (non-Javadoc)
   *
   * @seecom.sysmopee.core.model.servicemodel.IServiceModelAdapter#
   * getAddressableSize()
   */
  @Override
  public int getAddressableSize() {
    return 3;
  }

  /*
   * (non-Javadoc)
   *
   * @see
   * com.sysmopee.core.model.servicemodel.IServiceModelAdapter#getMemoryBlock
   * (int, int)
   */
  @Override
  public byte[] getMemoryBlock(int intValue, int length) {

    IMemory mem = fModel.getState().getMemories().get(0);

    byte[] res = new byte[length * 3];

    int max = Math.min(intValue + length, mem.getElements().length)
        - intValue;

    for (int i = 0; i < max; i++) {

      int val = ((IDataWord) mem.getElement(intValue + i))
          .getIntegerValue();

      res[i + 0] = (byte) (val & 0xFF);
      res[i + 1] = (byte) ((val & 0xFF00) >> 8);
      res[i + 2] = (byte) ((val & 0xFF0000) >> 16);
    }

    return res;
  }

  /*
   * (non-Javadoc)
   *
   * @seecom.sysmopee.core.model.servicemodel.IServiceModelAdapter#
   * getMemoryEndAddress()
   */
  @Override
  public int getMemoryEndAddress() {
    return fModel.getState().getMemories().get(0).getElements().length;
  }

  /*
```

```
 * (non-Javadoc)
 *
 * @see com.sysmopee.core.model.servicemodel.IServiceModelAdapter#
 * getMemoryStartAddress()
 */
@Override
public int getMemoryStartAddress() {
  return 0;
}

/*
 * (non-Javadoc)
 *
 * @see com.sysmopee.core.model.servicemodel.IServiceModelAdapter#
 * setValueOfMemoryLocation(java.math.BigInteger, byte[])
 */
@Override
public void setValueOfMemoryLocation(BigInteger add, byte[] bytes) {

}

@Override
public void initialize(ISimulator simulator) {
  // Get handles for the service request places of the model
  p0 = (IServiceRequestPlace) ((IHCPNServiceModel) fModel)
      .getPlace("P0");
  p1 = (IServiceRequestPlace) ((IHCPNServiceModel) fModel)
      .getPlace("P1");
  p2 = (IServiceRequestPlace) ((IHCPNServiceModel) fModel)
      .getPlace("P2");
}

abstract class SVFServiceRequest extends AbstractServiceRequest
    implements ICPUServiceRequest {

  private final IDataWord fProgramCounter;

  public SVFServiceRequest(IServiceModel model, IService type) {
    super(model, type);

    fProgramCounter = (IDataWord) model.getState()
        .getRegister("pc").getElement();
  }

  @Override
  public final void p0() {
    // Signal request
    this.requested();

    // Default action
    fProgramCounter.increment();

    // Produce
    p1.addServiceRequest(this);

    // Consume
    p0.removeServiceRequest();
  }

  public void p1b() {
    // Produce
    p2.addServiceRequest(this);

    // Consume
    p1.removeServiceRequest();
```

```
    }

    @Override
    public final void p2() {
      // Consume
      p2.removeServiceRequest();

      // Signal done
      this.done();
    }
  }

  abstract class EXTENDEDTYPE1 extends SVFServiceRequest {
    protected final IDataWord _GETOP1;
    protected final int _EXT1;
    protected final int _OP1;

    public EXTENDEDTYPE1(IServiceModel p0, IService p1, int OP1,
        int EXT1) {
      super(p0, p1);
      _OP1 = OP1;
      _EXT1 = EXT1;
      switch (_EXT1) {
      case 0:
        _GETOP1 = ((IDataWord) fModel.getState().getRegister(
            "reg" + _OP1).getElement());
        break;
      case 1:
        int v = _OP1 & 0x7;
        _GETOP1 = ((IDataWord) fModel.getState().getRegister(
            "cfgst" + v).getElement());
        break;
      case 2:
        _GETOP1 = ((IDataWord) fModel.getState().getRegister(
            "a_lp" + _OP1).getElement());
        break;
      case 3:
        _GETOP1 = ((IDataWord) fModel.getState().getRegister(
            "a_bp" + _OP1).getElement());
        break;
      case 4:
        _GETOP1 = ((IDataWord) fModel.getState().getRegister(
            "a_hp" + _OP1).getElement());
        break;
      case 5:
        _GETOP1 = ((IDataWord) fModel.getState().getRegister(
            "b_lp" + _OP1).getElement());
        break;
      case 6:
        _GETOP1 = ((IDataWord) fModel.getState().getRegister(
            "b_bp" + _OP1).getElement());
        break;
      default:
        _GETOP1 = ((IDataWord) fModel.getState().getRegister(
            "b_hp" + _OP1).getElement());
        break;
      }
    }
  }

  abstract class EXTENDEDTYPE2 extends SVFServiceRequest {
    protected final IDataWord _GETOP1;
    protected final IDataWord _GETOP2;
    protected final int _EXT1;
    protected final int _EXT2;
```

```
protected final int _OP1;
protected final int _OP2;

public EXTENDEDTYPE2(IServiceModel p0, IService p1, int OP2,
    int OP1, int EXT2, int EXT1) {
  super(p0, p1);
  _OP2 = OP2;
  _OP1 = OP1;
  _EXT2 = EXT2;
  _EXT1 = EXT1;
  switch (_EXT2) {
  case 0:
    _GETOP2 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP2).getElement());
    break;
  case 1:
    _GETOP2 = ((IDataWord) fModel.getState().getRegister(
        "cfgst" + _OP2).getElement());
    break;
  case 2:
    _GETOP2 = ((IDataWord) fModel.getState().getRegister(
        "a_lp" + _OP2).getElement());
    break;
  case 3:
    _GETOP2 = ((IDataWord) fModel.getState().getRegister(
        "a_bp" + _OP2).getElement());
    break;
  case 4:
    _GETOP2 = ((IDataWord) fModel.getState().getRegister(
        "a_hp" + _OP2).getElement());
    break;
  case 5:
    _GETOP2 = ((IDataWord) fModel.getState().getRegister(
        "b_lp" + _OP2).getElement());
    break;
  case 6:
    _GETOP2 = ((IDataWord) fModel.getState().getRegister(
        "b_bp" + _OP2).getElement());
    break;
  default:
    _GETOP2 = ((IDataWord) fModel.getState().getRegister(
        "b_hp" + _OP2).getElement());
    break;
  }
  switch (_EXT1) {
  case 0:
    _GETOP1 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP1).getElement());
    break;
  case 1:
    _GETOP1 = ((IDataWord) fModel.getState().getRegister(
        "cfgst" + _OP1).getElement());
    break;
  case 2:
    _GETOP1 = ((IDataWord) fModel.getState().getRegister(
        "a_lp" + _OP1).getElement());
    break;
  case 3:
    _GETOP1 = ((IDataWord) fModel.getState().getRegister(
        "a_bp" + _OP1).getElement());
    break;
  case 4:
    _GETOP1 = ((IDataWord) fModel.getState().getRegister(
        "a_hp" + _OP1).getElement());
    break;
```

```
      case 5:
        _GETOP1 = ((IDataWord) fModel.getState().getRegister(
            "b_lp" + _OP1).getElement());
        break;
      case 6:
        _GETOP1 = ((IDataWord) fModel.getState().getRegister(
            "b_bp" + _OP1).getElement());
        break;
      default:
        _GETOP1 = ((IDataWord) fModel.getState().getRegister(
            "b_hp" + _OP1).getElement());
        break;
      }
    }
  }

  abstract class EEXTENDEDTYPE1 extends EXTENDEDTYPE1 {
    protected final IDataWord _GETEOP1;
    protected final int _EXT1;
    protected final int _EEXT1;
    protected final int _OP1;

    public EEXTENDEDTYPE1(IServiceModel p0, IService p1, int OP1,
        int EEXT1, int EXT1) {
      super(p0, p1, OP1, EXT1);
      _OP1 = OP1;
      _EEXT1 = EEXT1;
      _EXT1 = EXT1;
      if ((_EEXT1 == 1)) {
        switch (_EXT1) {
        case 0:
          _GETEOP1 = ((IDataWord) fModel.getState().getRegister(
              "c1" + _OP1).getElement());
          break;
        case 1:
          _GETEOP1 = ((IDataWord) fModel.getState().getRegister(
              "c2" + _OP1).getElement());
          break;
        default:
          _GETEOP1 = ((IDataWord) fModel.getState().getRegister(
              "fcfg" + _OP1).getElement());
          break;
        }
      } else {
        _GETEOP1 = _GETOP1;
      }
    }
  }

  abstract class EEXTENDEDTYPE2 extends EXTENDEDTYPE2 {
    protected final IDataWord _GETEOP1;
    protected final IDataWord _GETEOP2;
    protected final int _EXT1;
    protected final int _EXT2;
    protected final int _EEXT1;
    protected final int _EEXT2;
    protected final int _OP1;

    public EEXTENDEDTYPE2(IServiceModel p0, IService p1, int OP2,
        int OP1, int EEXT2, int EEXT1, int EXT2, int EXT1) {
      super(p0, p1, OP2, OP1, EXT2, EXT1);
      _OP1 = OP1;
      _EEXT2 = EEXT2;
      _EEXT1 = EEXT1;
      _EXT2 = EXT2;
```

```
      _EXT1 = EXT1;
      if ((_EEXT2 == 1)) {
        switch (_EXT2) {
        case 0:
          _GETEOP2 = ((IDataWord) fModel.getState().getRegister(
              "c1" + _OP1).getElement());
          break;
        case 1:
          _GETEOP2 = ((IDataWord) fModel.getState().getRegister(
              "c2" + _OP1).getElement());
          break;
        default:
          _GETEOP2 = ((IDataWord) fModel.getState().getRegister(
              "fcfg" + _OP1).getElement());
          break;
        }
      } else {
        _GETEOP2 = _GETOP2;
      }
      if ((_EEXT1 == 1)) {
        switch (_EXT1) {
        case 0:
          _GETEOP1 = ((IDataWord) fModel.getState().getRegister(
              "c1" + _OP1).getElement());
          break;
        case 1:
          _GETEOP1 = ((IDataWord) fModel.getState().getRegister(
              "c2" + _OP1).getElement());
          break;
        default:
          _GETEOP1 = ((IDataWord) fModel.getState().getRegister(
              "fcfg" + _OP1).getElement());
          break;
        }
      } else {
        _GETEOP1 = _GETOP1;
      }
    }
  }

  class NOP extends SVFServiceRequest {
    public NOP(IServiceModel p0, IService p1) {
      super(p0, p1);
    }

    public final void p1() {
    }
  }

  abstract class TYPE1 extends EXTENDEDTYPE2 {
    public TYPE1(IServiceModel p0, IService p1, int OP2, int OP1,
        int EXT2, int EXT1) {
      super(p0, p1, OP2, OP1, EXT2, EXT1);
    }

    public abstract void p1();
  }

  class ADDS extends TYPE1 {
    protected final IDataWord _FIELD0;
    protected final int _OP3;

    public ADDS(IServiceModel p0, IService p1, int OP3, int OP2,
        int OP1, int EXT2, int EXT1) {
      super(p0, p1, OP2, OP1, EXT2, EXT1);
```

```
    _OP3 = OP3;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP3).getElement());
  }

  public final void p1() {
    _GETOP1.setIntegerValue((_GETOP2.getIntegerValue() + _FIELD0
        .getIntegerValue()));
  }
}

class ADDU extends TYPE1 {
  protected final IDataWord _FIELD0;
  protected final int _OP3;

  public ADDU(IServiceModel p0, IService p1, int OP3, int OP2,
      int OP1, int EXT2, int EXT1) {
    super(p0, p1, OP2, OP1, EXT2, EXT1);
    _OP3 = OP3;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP3).getElement());
  }

  public final void p1() {
    _GETOP1.setIntegerValue((_GETOP2.getIntegerValue() + _FIELD0
        .getIntegerValue()));
  }
}

class MULS extends TYPE1 {
  protected final IDataWord _FIELD0;
  protected final int _OP3;

  public MULS(IServiceModel p0, IService p1, int OP3, int OP2,
      int OP1, int EXT2, int EXT1) {
    super(p0, p1, OP2, OP1, EXT2, EXT1);
    _OP3 = OP3;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP3).getElement());
  }

  public final void p1() {
    _GETOP1.setIntegerValue((_GETOP2.getIntegerValue() * _FIELD0
        .getIntegerValue()));
  }
}

class MULU extends TYPE1 {
  protected final IDataWord _FIELD0;
  protected final int _OP3;

  public MULU(IServiceModel p0, IService p1, int OP3, int OP2,
      int OP1, int EXT2, int EXT1) {
    super(p0, p1, OP2, OP1, EXT2, EXT1);
    _OP3 = OP3;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP3).getElement());
  }

  public final void p1() {
    _GETOP1.setIntegerValue((_GETOP2.getIntegerValue() * _FIELD0
        .getIntegerValue()));
  }
}
```

```
class MACS extends TYPE1 {
  protected final IDataWord _FIELD0;
  protected final int _OP3;

  public MACS(IServiceModel p0, IService p1, int OP3, int OP2,
      int OP1, int EXT2, int EXT1) {
    super(p0, p1, OP2, OP1, EXT2, EXT1);
    _OP3 = OP3;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP3).getElement());
  }

  public final void p1() {
    _GETOP1.setIntegerValue(((_GETOP2.getIntegerValue() * _FIELD0
        .getIntegerValue()) + _GETOP1.getIntegerValue()));
  }
}

class MACU extends TYPE1 {
  protected final IDataWord _FIELD0;
  protected final int _OP3;

  public MACU(IServiceModel p0, IService p1, int OP3, int OP2,
      int OP1, int EXT2, int EXT1) {
    super(p0, p1, OP2, OP1, EXT2, EXT1);
    _OP3 = OP3;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP3).getElement());
  }

  public final void p1() {
    _GETOP1.setIntegerValue(((_GETOP2.getIntegerValue() * _FIELD0
        .getIntegerValue()) + _GETOP1.getIntegerValue()));
  }
}

class AND extends TYPE1 {
  protected final IDataWord _FIELD0;
  protected final int _OP3;

  public AND(IServiceModel p0, IService p1, int OP3, int OP2,
      int OP1, int EXT2, int EXT1) {
    super(p0, p1, OP2, OP1, EXT2, EXT1);
    _OP3 = OP3;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP3).getElement());
  }

  public final void p1() {
    _GETOP1.setIntegerValue((_GETOP2.getIntegerValue() & _FIELD0
        .getIntegerValue()));
  }
}

class OR extends TYPE1 {
  protected final IDataWord _FIELD0;
  protected final int _OP3;

  public OR(IServiceModel p0, IService p1, int OP3, int OP2, int OP1,
      int EXT2, int EXT1) {
    super(p0, p1, OP2, OP1, EXT2, EXT1);
    _OP3 = OP3;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP3).getElement());
  }
```

```
  public final void p1() {
    _GETOP1.setIntegerValue((_GETOP2.getIntegerValue() | _FIELD0
        .getIntegerValue()));
  }
}

class XOR extends TYPE1 {
  protected final IDataWord _FIELD0;
  protected final int _OP3;

  public XOR(IServiceModel p0, IService p1, int OP3, int OP2,
      int OP1, int EXT2, int EXT1) {
    super(p0, p1, OP2, OP1, EXT2, EXT1);
    _OP3 = OP3;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP3).getElement());
  }

  public final void p1() {
    _GETOP1.setIntegerValue((_GETOP2.getIntegerValue() ^ _FIELD0
        .getIntegerValue()));
  }
}

class LW extends TYPE1 {
  protected final IServiceRequest _FIELD10;
  protected final IDataWord _FIELD9;
  protected final IDataWord _FIELD8;
  protected final IServiceRequest _FIELD7;
  protected final IDataWord _FIELD6;
  protected final IDataWord _FIELD5;
  protected final IServiceRequest _FIELD4;
  protected final IDataWord _FIELD3;
  protected final IDataWord _FIELD2;
  protected final IDataWord[] _FIELD1;
  protected final IDataWord _FIELD0;
  protected final IServiceModelInterface _SOURCE_LEFT;
  protected final IServiceModelInterface _SOURCE_RIGHT;
  protected final IServiceModelInterface _I2C;
  protected final int _OP3;

  public LW(IServiceModel p0, IService p1, int OP3, int OP2, int OP1,
      int EXT2, int EXT1) {
    super(p0, p1, OP2, OP1, EXT2, EXT1);
    _OP3 = OP3;
    _I2C = p0.getActiveServiceModelInterface("I2C");
    _SOURCE_LEFT = p0.getActiveServiceModelInterface("SOURCE_LEFT");
    _SOURCE_RIGHT = p0
        .getActiveServiceModelInterface("SOURCE_RIGHT");
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP3).getElement());
    _FIELD1 = (IDataWord[]) fModel.getState().getMemorySegment(
        "mem").getElements();
    _FIELD2 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP3).getElement());
    _FIELD3 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP3).getElement());
    _FIELD4 = _SOURCE_LEFT.createServiceRequest("READ",
        new Object[] { _GETOP1 });
    _FIELD5 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP3).getElement());
    _FIELD6 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP3).getElement());
    _FIELD8 = new DataWord(24);
```

```
    _FIELD9 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP3).getElement());
    _FIELD7 = _I2C.createServiceRequest("READ", new Object[] {
        _GETOP1, _FIELD8 });
    if (_SOURCE_RIGHT != null)
      _FIELD10 = _SOURCE_RIGHT.createServiceRequest("READ",
          new Object[] { _GETOP1 });
    else
      _FIELD10 = null;
  }

  public final void p1() {
    if (((_GETOP2.getIntegerValue()
      + _FIELD0.getIntegerValue()) < 512)) {
      _GETOP1
          .setIntegerValue(_FIELD1[(_GETOP2.getIntegerValue() +
          _FIELD2.getIntegerValue())].getIntegerValue());
    } else if (((_GETOP2.getIntegerValue() + _FIELD3
        .getIntegerValue()) == 2048)) {
      _SOURCE_LEFT.request(_FIELD4);

      _FIELD4.addDoneListener(new IServiceDoneListener() {

        @Override
        public void done() {
          setActive(true);
          p1b();
        }

      });

      setActive(false);
    } else if (((_GETOP2.getIntegerValue() + _FIELD3
        .getIntegerValue()) == 2049)) {
      _SOURCE_RIGHT.request(_FIELD10);

      _FIELD10.addDoneListener(new IServiceDoneListener() {

        @Override
        public void done() {
          setActive(true);
          p1b();
        }
      });

      setActive(false);
    } else if ((((_GETOP2.getIntegerValue() + _FIELD5
        .getIntegerValue()) >= 2560) && ((_GETOP2
        .getIntegerValue() + _FIELD6.getIntegerValue()) < 2592))) {
      _FIELD8
          .setIntegerValue((int) (((_GETOP2.getIntegerValue() +
              _FIELD9.getIntegerValue()) & 31)));
      _I2C.request(_FIELD7);
    }
  }
}

class SW extends TYPE1 {
  protected final IServiceRequest _FIELD10;
  protected final IDataWord _FIELD9;
  protected final IDataWord _FIELD8;
  protected final IServiceRequest _FIELD7;
  protected final IDataWord _FIELD6;
  protected final IDataWord _FIELD5;
  protected final IServiceRequest _FIELD4;
```

```
  protected final IDataWord _FIELD3;
  protected final IDataWord _FIELD2;
  protected final IDataWord[] _FIELD1;
  protected final IDataWord _FIELD0;
  protected final IServiceModelInterface _I2C;
  protected final IServiceModelInterface _SINK_LEFT;
  protected final IServiceModelInterface _SINK_RIGHT;
  protected final int _OP3;

  public SW(IServiceModel p0, IService p1, int OP3, int OP2, int OP1,
      int EXT2, int EXT1) {
    super(p0, p1, OP2, OP1, EXT2, EXT1);
    _OP3 = OP3;
    _SINK_LEFT = p0.getActiveServiceModelInterface("SINK_LEFT");
    _SINK_RIGHT = p0.getActiveServiceModelInterface("SINK_RIGHT");
    _I2C = p0.getActiveServiceModelInterface("I2C");
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP3).getElement());
    _FIELD1 = (IDataWord[]) fModel.getState().getMemorySegment(
        "mem").getElements();
    _FIELD2 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP3).getElement());
    _FIELD3 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP3).getElement());
    _FIELD4 = _SINK_LEFT.createServiceRequest("WRITE",
        new Object[] { _GETOP1 });
    _FIELD5 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP3).getElement());
    _FIELD6 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP3).getElement());
    _FIELD8 = new DataWord(24);
    _FIELD9 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP3).getElement());
    _FIELD7 = _I2C.createServiceRequest("WRITE", new Object[] {
        _GETOP1, _FIELD8 });
    _FIELD10 = _SINK_RIGHT.createServiceRequest("WRITE",
        new Object[] { _GETOP1 });
  }

  public final void p1() {
    if (((_GETOP2.getIntegerValue() +
      _FIELD0.getIntegerValue()) < 512)) {
      _FIELD1[(_GETOP2.getIntegerValue() + _FIELD2
          .getIntegerValue())].setIntegerValue(_GETOP1
          .getIntegerValue());
    } else if (((_GETOP2.getIntegerValue() + _FIELD3
        .getIntegerValue()) == 2304)) {
      _SINK_LEFT.request(_FIELD4);
    } else if (((_GETOP2.getIntegerValue() + _FIELD3
        .getIntegerValue()) == 2305)) {
      _SINK_RIGHT.request(_FIELD10);
    } else if ((((_GETOP2.getIntegerValue() + _FIELD5
        .getIntegerValue()) >= 2560) && ((_GETOP2
        .getIntegerValue() + _FIELD6.getIntegerValue()) < 2592))) {
      _FIELD8
          .setIntegerValue((int) (((_GETOP2.getIntegerValue() + _FIELD9
              .getIntegerValue()) & 31)));
      _I2C.request(_FIELD7);
    }
  }
}

abstract class TYPE2 extends SVFServiceRequest {
  public TYPE2(IServiceModel p0, IService p1) {
    super(p0, p1);
```

```
  }

  public abstract void p1();
}

class ADDIS extends TYPE2 {
  protected final int _FIELD2;
  protected final IDataWord _FIELD1;
  protected final IDataWord _FIELD0;
  protected final int _IMMEEXT;
  protected final int _OP1;
  protected final int _OP2;

  public ADDIS(IServiceModel p0, IService p1, int OP2, int OP1,
      int IMMEEXT) {
    super(p0, p1);
    _OP2 = OP2;
    _OP1 = OP1;
    _IMMEEXT = IMMEEXT;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP1).getElement());
    _FIELD1 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP2).getElement());
    _FIELD2 = _IMMEEXT >= 2048 ? _IMMEEXT - 4096 : _IMMEEXT;
  }

  public final void p1() {
    _FIELD0.setIntegerValue((_FIELD0.getIntegerValue() + _FIELD2));
  }
}

class ADDIU extends TYPE2 {
  protected final IDataWord _FIELD1;
  protected final IDataWord _FIELD0;
  protected final int _IMMEEXT;
  protected final int _OP1;
  protected final int _OP2;

  public ADDIU(IServiceModel p0, IService p1, int OP2, int OP1,
      int IMMEEXT) {
    super(p0, p1);
    _OP2 = OP2;
    _OP1 = OP1;
    _IMMEEXT = IMMEEXT;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP1).getElement());
    _FIELD1 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP2).getElement());
  }

  public final void p1() {
    _FIELD0.setIntegerValue((_FIELD0.getIntegerValue() + _IMMEEXT));
  }
}

class MULIS extends TYPE2 {
  protected final int _FIELD2;
  protected final IDataWord _FIELD1;
  protected final IDataWord _FIELD0;
  protected final int _IMMEEXT;
  protected final int _OP1;
  protected final int _OP2;

  public MULIS(IServiceModel p0, IService p1, int OP2, int OP1,
      int IMMEEXT) {
```

```
    super(p0, p1);
    _OP2 = OP2;
    _OP1 = OP1;
    _IMMEEXT = IMMEEXT;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP1).getElement());
    _FIELD1 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP2).getElement());
    _FIELD2 = _IMMEEXT >= 2048 ? _IMMEEXT - 4096 : _IMMEEXT;
  }

  public final void p1() {
    _FIELD0.setIntegerValue((_FIELD0.getIntegerValue() * _FIELD2));
  }
}

class MULIU extends TYPE2 {
  protected final IDataWord _FIELD1;
  protected final IDataWord _FIELD0;
  protected final int _IMMEEXT;
  protected final int _OP1;
  protected final int _OP2;

  public MULIU(IServiceModel p0, IService p1, int OP2, int OP1,
      int IMMEEXT) {
    super(p0, p1);
    _OP2 = OP2;
    _OP1 = OP1;
    _IMMEEXT = IMMEEXT;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP1).getElement());
    _FIELD1 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP2).getElement());
  }

  public final void p1() {
    _FIELD0.setIntegerValue((_FIELD0.getIntegerValue() * _IMMEEXT));
  }
}

class MOVI extends TYPE2 {
  protected final int _FIELD2;
  protected final IDataWord _FIELD1;
  protected final IDataWord _FIELD0;
  protected final int _IMMEEXT;
  protected final int _OP1;

  public MOVI(IServiceModel p0, IService p1, int OP1, int IMMEEXT) {
    super(p0, p1);
    _OP1 = OP1;
    _IMMEEXT = IMMEEXT;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP1).getElement());
    _FIELD1 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP1).getElement());
    _FIELD2 = (_IMMEEXT & 4095);
  }

  public final void p1() {
    _FIELD0
        .setIntegerValue((int) (((((_FIELD1.getIntegerValue()
        & 16773120) >>> 12) & 16773120) | _FIELD2)));
  }
}
```

```
class MOVUI extends TYPE2 {
  protected final IDataWord _FIELD3;
  protected final int _FIELD2;
  protected final int _FIELD1;
  protected final IDataWord _FIELD0;
  protected final int _IMMEEXT;
  protected final int _OP1;

  public MOVUI(IServiceModel p0, IService p1, int OP1, int IMMEEXT) {
    super(p0, p1);
    _OP1 = OP1;
    _IMMEEXT = IMMEEXT;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP1).getElement());
    _FIELD1 = (_IMMEEXT & 4095);
    _FIELD2 = (_FIELD1 << 12);
    _FIELD3 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP1).getElement());
  }

  public final void p1() {
    _FIELD0.setIntegerValue((int) ((_FIELD2 | (_FIELD3
        .getIntegerValue() & 4095))));
  }
}

class LWA extends TYPE2 {
  protected final IServiceRequest _FIELD7;
  protected final IDataWord _FIELD6;
  protected final IServiceRequest _FIELD5;
  protected final IDataWord _FIELD4;
  protected final IServiceRequest _FIELD3;
  protected final IDataWord _FIELD2;
  protected final IDataWord _FIELD1;
  protected final IEvaluate _FIELD0;
  protected final IServiceModelInterface _SOURCE_LEFT;
  protected final IServiceModelInterface _SOURCE_RIGHT;
  protected final IServiceModelInterface _I2C;
  protected final int _OP1;
  protected final int _IMMEEXT;

  public LWA(IServiceModel p0, IService p1, int IMMEEXT, int OP1) {
    super(p0, p1);
    _IMMEEXT = IMMEEXT;
    _OP1 = OP1;
    _I2C = p0.getActiveServiceModelInterface("I2C");
    _SOURCE_LEFT = p0.getActiveServiceModelInterface("SOURCE_LEFT");
    _SOURCE_RIGHT = p0
        .getActiveServiceModelInterface("SOURCE_RIGHT");
    _FIELD1 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP1).getElement());
    _FIELD2 = ((IDataWord) fModel.getState().getMemoryLocation(
        _IMMEEXT));
    if ((_IMMEEXT < 512)) {
      _FIELD0 = new IEvaluate() {
        public final void evaluate() {
          _FIELD1.setIntegerValue(_FIELD2.getIntegerValue());
        }
      };
    } else if ((_IMMEEXT == 2048)) {
      _FIELD0 = new IEvaluate() {
        public final void evaluate() {
          _SOURCE_LEFT.request(_FIELD3);

          _FIELD3.addDoneListener(new IServiceDoneListener() {
```

```
            @Override
            public void done() {
              setActive(true);
              p1b();
            }

          });

          setActive(false);
        }
      };
    } else if ((_IMMEEXT == 2049)) {
      _FIELD0 = new IEvaluate() {
        public final void evaluate() {
          _SOURCE_RIGHT.request(_FIELD7);

          /*
           * _FIELD7.addDoneListener(new
           * IServiceDoneListener(){
           *
           * @Override public void done() { setActive(true);
           * p1b(); }
           *
           * });
           *
           * setActive(false);
           */
        }
      };
    } else if (((_IMMEEXT >= 2560) && (_IMMEEXT < 2592))) {
      _FIELD0 = new IEvaluate() {
        public final void evaluate() {
          _I2C.request(_FIELD5);
        }
      };
    } else {
      _FIELD0 = new IEvaluate() {
        public final void evaluate() {
          {
          }
        }
      };
    }
    _FIELD4 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP1).getElement());
    _FIELD3 = _SOURCE_LEFT.createServiceRequest("READ",
        new Object[] { _FIELD4 });
    _FIELD6 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP1).getElement());
    _FIELD5 = _I2C.createServiceRequest("READ", new Object[] {
        _FIELD6, new DataWord(_IMMEEXT & 31, 24) });
    if (_SOURCE_RIGHT != null)
      _FIELD7 = _SOURCE_RIGHT.createServiceRequest("READ",
          new Object[] { _FIELD4 });
    else
      _FIELD7 = null;
  }

  public final void p1() {
    _FIELD0.evaluate();
  }
}

class SWA extends TYPE2 {
```

```
protected final IServiceRequest _FIELD7;
protected final IDataWord _FIELD6;
protected final IServiceRequest _FIELD5;
protected final IDataWord _FIELD4;
protected final IServiceRequest _FIELD3;
protected final IDataWord _FIELD2;
protected final IDataWord _FIELD1;
protected final IEvaluate _FIELD0;
protected final IServiceModelInterface _I2C;
protected final IServiceModelInterface _SINK_LEFT;
protected final IServiceModelInterface _SINK_RIGHT;
protected final int _OP1;
protected final int _IMMEEXT;

public SWA(IServiceModel p0, IService p1, int IMMEEXT, int OP1) {
  super(p0, p1);
  _IMMEEXT = IMMEEXT;
  _OP1 = OP1;
  _SINK_LEFT = p0.getActiveServiceModelInterface("SINK_LEFT");
  _SINK_RIGHT = p0.getActiveServiceModelInterface("SINK_RIGHT");
  _I2C = p0.getActiveServiceModelInterface("I2C");
  _FIELD1 = ((IDataWord) fModel.getState().getMappedElement(
      _IMMEEXT));
  _FIELD2 = ((IDataWord) fModel.getState().getRegister(
      "reg" + _OP1).getElement());
  if ((_IMMEEXT < 512)) {
    _FIELD0 = new IEvaluate() {
      public final void evaluate() {
        _FIELD1.setIntegerValue(_FIELD2.getIntegerValue());
      }
    };
  } else if ((_IMMEEXT == 2304)) {
    _FIELD0 = new IEvaluate() {
      public final void evaluate() {
        _SINK_LEFT.request(_FIELD3);
      }
    };
  } else if ((_IMMEEXT == 2305)) {
    _FIELD0 = new IEvaluate() {
      public final void evaluate() {
        _SINK_RIGHT.request(_FIELD7);
      }
    };
  } else if (((_IMMEEXT >= 2560) && (_IMMEEXT < 2592))) {
    _FIELD0 = new IEvaluate() {
      public final void evaluate() {
        _I2C.request(_FIELD5);
      }
    };
  } else {
    _FIELD0 = new IEvaluate() {
      public final void evaluate() {
        {
        }
      }
    };
  }
  _FIELD4 = ((IDataWord) fModel.getState().getRegister(
      "reg" + _OP1).getElement());
  _FIELD3 = _SINK_LEFT.createServiceRequest("WRITE",
      new Object[] { _FIELD4 });
  _FIELD6 = ((IDataWord) fModel.getState().getRegister(
      "reg" + _OP1).getElement());
  _FIELD5 = _I2C.createServiceRequest("WRITE", new Object[] {
      _FIELD6, new DataWord(_IMMEEXT & 31, 24) });
```

```
    if (_SINK_RIGHT != null)
      _FIELD7 = _SINK_RIGHT.createServiceRequest("WRITE",
          new Object[] { _FIELD4 });
    else
      _FIELD7 = null;
  }

  public final void p1() {
    _FIELD0.evaluate();
  }
}

class ANDI extends TYPE2 {
  protected final IDataWord _FIELD1;
  protected final IDataWord _FIELD0;
  protected final int _IMMEEXT;
  protected final int _OP1;
  protected final int _OP2;

  public ANDI(IServiceModel p0, IService p1, int OP2, int OP1,
      int IMMEEXT) {
    super(p0, p1);
    _OP2 = OP2;
    _OP1 = OP1;
    _IMMEEXT = IMMEEXT;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP1).getElement());
    _FIELD1 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP2).getElement());
  }

  public final void p1() {
    _FIELD0.setIntegerValue((_FIELD0.getIntegerValue() & _IMMEEXT));
  }
}

class ORI extends TYPE2 {
  protected final IDataWord _FIELD1;
  protected final IDataWord _FIELD0;
  protected final int _IMMEEXT;
  protected final int _OP1;
  protected final int _OP2;

  public ORI(IServiceModel p0, IService p1, int OP2, int OP1,
      int IMMEEXT) {
    super(p0, p1);
    _OP2 = OP2;
    _OP1 = OP1;
    _IMMEEXT = IMMEEXT;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP1).getElement());
    _FIELD1 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP2).getElement());
  }

  public final void p1() {
    _FIELD0.setIntegerValue((_FIELD0.getIntegerValue() | _IMMEEXT));
  }
}

class XORI extends TYPE2 {
  protected final IDataWord _FIELD1;
  protected final IDataWord _FIELD0;
  protected final int _IMMEEXT;
  protected final int _OP1;
```

```
    protected final int _OP2;

    public XORI(IServiceModel p0, IService p1, int OP2, int OP1,
        int IMMEEXT) {
      super(p0, p1);
      _OP2 = OP2;
      _OP1 = OP1;
      _IMMEEXT = IMMEEXT;
      _FIELD0 = ((IDataWord) fModel.getState().getRegister(
          "reg" + _OP1).getElement());
      _FIELD1 = ((IDataWord) fModel.getState().getRegister(
          "reg" + _OP2).getElement());
    }

    public final void p1() {
      _FIELD0.setIntegerValue((_FIELD0.getIntegerValue() ^ _IMMEEXT));
    }
  }

  class BGTZ extends TYPE2 {
    protected final int _FIELD3;
    protected final IDataWord _FIELD2;
    protected final IDataWord _FIELD1;
    protected final IDataWord _FIELD0;
    protected final int _IMMEEXT;
    protected final int _OP1;

    public BGTZ(IServiceModel p0, IService p1, int OP1, int IMMEEXT) {
      super(p0, p1);
      _OP1 = OP1;
      _IMMEEXT = IMMEEXT;
      _FIELD0 = ((IDataWord) fModel.getState().getRegister(
          "reg" + _OP1).getElement());
      _FIELD1 = (IDataWord) fModel.getState().getRegister("pc")
          .getElement();
      _FIELD2 = (IDataWord) fModel.getState().getRegister("pc")
          .getElement();
      _FIELD3 = _IMMEEXT >= 2048 ? _IMMEEXT - 4096 : _IMMEEXT;
    }

    public final void p1() {
      if ((_FIELD0.getIntegerValue() > 0)) {
        _FIELD1
            .setIntegerValue((_FIELD2.getIntegerValue() + _FIELD3));
      }
    }
  }

  class BLEZ extends TYPE2 {
    protected final int _FIELD3;
    protected final IDataWord _FIELD2;
    protected final IDataWord _FIELD1;
    protected final IDataWord _FIELD0;
    protected final int _IMMEEXT;
    protected final int _OP1;

    public BLEZ(IServiceModel p0, IService p1, int OP1, int IMMEEXT) {
      super(p0, p1);
      _OP1 = OP1;
      _IMMEEXT = IMMEEXT;
      _FIELD0 = ((IDataWord) fModel.getState().getRegister(
          "reg" + _OP1).getElement());
      _FIELD1 = (IDataWord) fModel.getState().getRegister("pc")
          .getElement();
      _FIELD2 = (IDataWord) fModel.getState().getRegister("pc")
```

```
        .getElement();
    _FIELD3 = _IMMEEXT >= 2048 ? _IMMEEXT - 4096 : _IMMEEXT;
  }

  public final void p1() {
    if ((_FIELD0.getIntegerValue() <= 0)) {
      _FIELD1
          .setIntegerValue((_FIELD2.getIntegerValue() + _FIELD3));
    }
  }
}

abstract class TYPE3 extends SVFServiceRequest {
  public TYPE3(IServiceModel p0, IService p1) {
    super(p0, p1);
  }

  public abstract void p1();
}

class JMP extends TYPE3 {
  protected final IDataWord _FIELD0;
  protected final int _TARGET;

  public JMP(IServiceModel p0, IService p1, int TARGET) {
    super(p0, p1);
    _TARGET = TARGET;
    _FIELD0 = (IDataWord) fModel.getState().getRegister("pc")
        .getElement();
  }

  public final void p1() {
    _FIELD0.setIntegerValue((int) (_TARGET));
  }
}

abstract class TYPE4 extends EXTENDEDTYPE2 {
  public TYPE4(IServiceModel p0, IService p1, int OP2, int OP1,
      int EXT2, int EXT1) {
    super(p0, p1, OP2, OP1, EXT2, EXT1);
  }

  public abstract void p1();
}

class SVF1 extends TYPE4 {
  protected final IDataWord _FIELD3;
  protected final IDataWord _FIELD2;
  protected final IDataWord _FIELD1;
  protected final IDataWord _FIELD0;
  protected final int _OP2;

  public SVF1(IServiceModel p0, IService p1, int OP2, int OP1,
      int EXT2, int EXT1) {
    super(p0, p1, OP2, OP1, EXT2, EXT1);
    _OP2 = OP2;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "a_hp" + _OP2).getElement());
    _FIELD1 = ((IDataWord) fModel.getState().getRegister(
        "a_lp" + _OP2).getElement());
    _FIELD2 = ((IDataWord) fModel.getState().getRegister(
        "c1" + _OP2).getElement());
    _FIELD3 = ((IDataWord) fModel.getState().getRegister(
        "a_lp" + _OP2).getElement());
  }
```

```
  public final void p1() {
    _FIELD0
        .setIntegerValue((((_GETOP1.getIntegerValue() - _FIELD1
            .getIntegerValue()) * _FIELD2.getIntegerValue()) + _FIELD3
            .getIntegerValue()));
  }
}

class SVF2A extends TYPE4 {
  protected final IDataWord _FIELD3;
  protected final IDataWord _FIELD2;
  protected final IDataWord _FIELD1;
  protected final IDataWord _FIELD0;
  protected final int _OP2;

  public SVF2A(IServiceModel p0, IService p1, int OP2, int OP1,
      int EXT2, int EXT1) {
    super(p0, p1, OP2, OP1, EXT2, EXT1);
    _OP2 = OP2;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "a_hp" + _OP2).getElement());
    _FIELD1 = ((IDataWord) fModel.getState().getRegister(
        "a_lp" + _OP2).getElement());
    _FIELD2 = ((IDataWord) fModel.getState().getRegister(
        "c1" + _OP2).getElement());
    _FIELD3 = ((IDataWord) fModel.getState().getRegister(
        "a_lp" + _OP2).getElement());
  }

  public final void p1() {
    _FIELD0
        .setIntegerValue((((_GETOP1.getIntegerValue() - _FIELD1
            .getIntegerValue()) * _FIELD2.getIntegerValue()) + _FIELD3
            .getIntegerValue()));
  }
}

class SVF2B extends TYPE4 {
  protected final IDataWord _FIELD3;
  protected final IDataWord _FIELD2;
  protected final IDataWord _FIELD1;
  protected final IDataWord _FIELD0;
  protected final int _OP2;

  public SVF2B(IServiceModel p0, IService p1, int OP2, int OP1,
      int EXT2, int EXT1) {
    super(p0, p1, OP2, OP1, EXT2, EXT1);
    _OP2 = OP2;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "a_hp" + _OP2).getElement());
    _FIELD1 = ((IDataWord) fModel.getState().getRegister(
        "a_lp" + _OP2).getElement());
    _FIELD2 = ((IDataWord) fModel.getState().getRegister(
        "c1" + _OP2).getElement());
    _FIELD3 = ((IDataWord) fModel.getState().getRegister(
        "a_lp" + _OP2).getElement());
  }

  public final void p1() {
    _FIELD0
        .setIntegerValue((((_GETOP1.getIntegerValue() - _FIELD1
            .getIntegerValue()) * _FIELD2.getIntegerValue()) + _FIELD3
            .getIntegerValue()));
  }
```

```
}

abstract class TYPE5 extends EXTENDEDTYPE1 {
  public TYPE5(IServiceModel p0, IService p1, int OP1, int EXT1) {
    super(p0, p1, OP1, EXT1);
  }

  public abstract void p1();
}

class SLLI extends TYPE5 {
  protected final IDataWord _FIELD0;
  protected final int _IMMEXT;
  protected final int _OP2;

  public SLLI(IServiceModel p0, IService p1, int OP2, int OP1,
      int IMMEXT, int EXT1) {
    super(p0, p1, OP1, EXT1);
    _OP2 = OP2;
    _IMMEXT = IMMEXT;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP2).getElement());
  }

  public final void p1() {
    _GETOP1.setIntegerValue((_FIELD0.getIntegerValue() << _IMMEXT));
  }
}

class SRLI extends TYPE5 {
  protected final IDataWord _FIELD0;
  protected final int _IMMEXT;
  protected final int _OP2;

  public SRLI(IServiceModel p0, IService p1, int OP2, int OP1,
      int IMMEXT, int EXT1) {
    super(p0, p1, OP1, EXT1);
    _OP2 = OP2;
    _IMMEXT = IMMEXT;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP2).getElement());
  }

  public final void p1() {
    _GETOP1
        .setIntegerValue((_FIELD0.getIntegerValue() >>> _IMMEXT));
  }
}

class SRAI extends TYPE5 {
  protected final IDataWord _FIELD0;
  protected final int _IMMEXT;
  protected final int _OP2;

  public SRAI(IServiceModel p0, IService p1, int OP2, int OP1,
      int IMMEXT, int EXT1) {
    super(p0, p1, OP1, EXT1);
    _OP2 = OP2;
    _IMMEXT = IMMEXT;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP2).getElement());
  }

  public final void p1() {
    _GETOP1.setIntegerValue((_FIELD0.getIntegerValue() >> _IMMEXT));
```

```
  }
}

class BGT extends TYPE5 {
  protected final int _FIELD3;
  protected final IDataWord _FIELD2;
  protected final IDataWord _FIELD1;
  protected final IDataWord _FIELD0;
  protected final int _IMMEXT;
  protected final int _OP2;

  public BGT(IServiceModel p0, IService p1, int OP2, int OP1,
      int IMMEXT, int EXT1) {
    super(p0, p1, OP1, EXT1);
    _OP2 = OP2;
    _IMMEXT = IMMEXT;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP2).getElement());
    _FIELD1 = (IDataWord) fModel.getState().getRegister("pc")
        .getElement();
    _FIELD2 = (IDataWord) fModel.getState().getRegister("pc")
        .getElement();
    _FIELD3 = _IMMEXT >= 64 ? _IMMEXT - 128 : _IMMEXT;
  }

  public final void p1() {
    if ((_GETOP1.getIntegerValue() > _FIELD0.getIntegerValue())) {
      _FIELD1
          .setIntegerValue((_FIELD2.getIntegerValue() + _FIELD3));
    }
  }
}

class BLT extends TYPE5 {
  protected final int _FIELD3;
  protected final IDataWord _FIELD2;
  protected final IDataWord _FIELD1;
  protected final IDataWord _FIELD0;
  protected final int _IMMEXT;
  protected final int _OP2;

  public BLT(IServiceModel p0, IService p1, int OP2, int OP1,
      int IMMEXT, int EXT1) {
    super(p0, p1, OP1, EXT1);
    _OP2 = OP2;
    _IMMEXT = IMMEXT;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP2).getElement());
    _FIELD1 = (IDataWord) fModel.getState().getRegister("pc")
        .getElement();
    _FIELD2 = (IDataWord) fModel.getState().getRegister("pc")
        .getElement();
    _FIELD3 = _IMMEXT >= 64 ? _IMMEXT - 128 : _IMMEXT;
  }

  public final void p1() {
    if ((_GETOP1.getIntegerValue() <= _FIELD0.getIntegerValue())) {
      _FIELD1
          .setIntegerValue((_FIELD2.getIntegerValue() + _FIELD3));
    }
  }
}

class BEQ extends TYPE5 {
  protected final int _FIELD3;
```

```
  protected final IDataWord _FIELD2;
  protected final IDataWord _FIELD1;
  protected final IDataWord _FIELD0;
  protected final int _IMMEXT;
  protected final int _OP2;

  public BEQ(IServiceModel p0, IService p1, int OP2, int OP1,
      int IMMEXT, int EXT1) {
    super(p0, p1, OP1, EXT1);
    _OP2 = OP2;
    _IMMEXT = IMMEXT;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP2).getElement());
    _FIELD1 = (IDataWord) fModel.getState().getRegister("pc")
        .getElement();
    _FIELD2 = (IDataWord) fModel.getState().getRegister("pc")
        .getElement();
    _FIELD3 = _IMMEXT >= 64 ? _IMMEXT - 128 : _IMMEXT;
  }

  public final void p1() {
    if ((_GETOP1.getIntegerValue() == _FIELD0.getIntegerValue())) {
      _FIELD1
          .setIntegerValue((_FIELD2.getIntegerValue() + _FIELD3));
    }
  }
}

class BNE extends TYPE5 {
  protected final int _FIELD3;
  protected final IDataWord _FIELD2;
  protected final IDataWord _FIELD1;
  protected final IDataWord _FIELD0;
  protected final int _IMMEXT;
  protected final int _OP2;

  public BNE(IServiceModel p0, IService p1, int OP2, int OP1,
      int IMMEXT, int EXT1) {
    super(p0, p1, OP1, EXT1);
    _OP2 = OP2;
    _IMMEXT = IMMEXT;
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP2).getElement());
    _FIELD1 = (IDataWord) fModel.getState().getRegister("pc")
        .getElement();
    _FIELD2 = (IDataWord) fModel.getState().getRegister("pc")
        .getElement();
    _FIELD3 = _IMMEXT >= 64 ? _IMMEXT - 128 : _IMMEXT;
  }

  public final void p1() {
    if ((_GETOP1.getIntegerValue() != _FIELD0.getIntegerValue())) {
      _FIELD1
          .setIntegerValue((_FIELD2.getIntegerValue() + _FIELD3));
    }
  }
}

abstract class TYPE6 extends EEXTENDEDTYPE2 {
  public TYPE6(IServiceModel p0, IService p1, int OP2, int OP1,
      int EEXT2, int EEXT1, int EXT2, int EXT1) {
    super(p0, p1, OP2, OP1, EEXT2, EEXT1, EXT2, EXT1);
  }

  public abstract void p1();
```

```
}

class MOV extends TYPE6 {
  public MOV(IServiceModel p0, IService p1, int OP2, int OP1,
      int EEXT2, int EEXT1, int EXT2, int EXT1) {
    super(p0, p1, OP2, OP1, EEXT2, EEXT1, EXT2, EXT1);
  }

  public final void p1() {
    _GETOP1.setIntegerValue(_GETOP2.getIntegerValue());
  }
}

class ABS extends TYPE6 {
  protected final IEvaluate _FIELD0;

  public ABS(IServiceModel p0, IService p1, int OP2, int OP1,
      int EEXT2, int EEXT1, int EXT2, int EXT1) {
    super(p0, p1, OP2, OP1, EEXT2, EEXT1, EXT2, EXT1);
    if ((_GETOP2.getIntegerValue() < 0)) {
      _FIELD0 = new IEvaluate() {
        public final void evaluate() {
          _GETOP1
              .setIntegerValue((_GETOP2.getIntegerValue() + 1));
        }
      };
    } else {
      _FIELD0 = new IEvaluate() {
        public final void evaluate() {
          {
          }
        }
      };
    }
  }

  public final void p1() {
    _FIELD0.evaluate();
  }
}

abstract class TYPE7 extends SVFServiceRequest {
  public TYPE7(IServiceModel p0, IService p1) {
    super(p0, p1);
  }

  public abstract void p1();
}

abstract class TYPE8 extends EEXTENDEDTYPE1 {
  public TYPE8(IServiceModel p0, IService p1, int OP1, int EEXT1,
      int EXT1) {
    super(p0, p1, OP1, EEXT1, EXT1);
  }

  public abstract void p1();
}

class LWI extends TYPE8 {
  protected final IServiceRequest _FIELD10;
  protected final IDataWord _FIELD9;
  protected final IDataWord _FIELD8;
  protected final IServiceRequest _FIELD7;
  protected final IDataWord _FIELD6;
  protected final IDataWord _FIELD5;
```

```
protected final IServiceRequest _FIELD4;
protected final IDataWord _FIELD3;
protected final IDataWord _FIELD2;
protected final IDataWord[] _FIELD1;
protected final IDataWord _FIELD0;
protected final IServiceModelInterface _SOURCE_LEFT;
protected final IServiceModelInterface _SOURCE_RIGHT;
protected final IServiceModelInterface _I2C;
protected final int _IMM;
protected final int _OP2;

public LWI(IServiceModel p0, IService p1, int OP2, int OP1,
    int IMM, int EEXT1, int EXT1) {
  super(p0, p1, OP1, EEXT1, EXT1);
  _OP2 = OP2;
  _IMM = IMM;
  _I2C = p0.getActiveServiceModelInterface("I2C");
  _SOURCE_LEFT = p0.getActiveServiceModelInterface("SOURCE_LEFT");
  _SOURCE_RIGHT = p0
      .getActiveServiceModelInterface("SOURCE_RIGHT");
  _FIELD0 = ((IDataWord) fModel.getState().getRegister(
      "reg" + _OP2).getElement());
  _FIELD1 = (IDataWord[]) fModel.getState().getMemorySegment(
      "mem").getElements();
  _FIELD2 = ((IDataWord) fModel.getState().getRegister(
      "reg" + _OP2).getElement());
  _FIELD3 = ((IDataWord) fModel.getState().getRegister(
      "reg" + _OP2).getElement());
  _FIELD4 = _SOURCE_LEFT.createServiceRequest("READ",
      new Object[] { _GETEOP1 });
  if (_SOURCE_RIGHT != null)
    _FIELD10 = _SOURCE_RIGHT.createServiceRequest("READ",
        new Object[] { _GETEOP1 });
  else
    _FIELD10 = null;
  _FIELD5 = ((IDataWord) fModel.getState().getRegister(
      "reg" + _OP2).getElement());
  _FIELD6 = ((IDataWord) fModel.getState().getRegister(
      "reg" + _OP2).getElement());
  _FIELD8 = new DataWord(24);
  _FIELD9 = ((IDataWord) fModel.getState().getRegister(
      "reg" + _OP2).getElement());
  _FIELD7 = _I2C.createServiceRequest("READ", new Object[] {
      _GETEOP1, _FIELD8 });
}

public final void p1() {
  if (((_FIELD0.getIntegerValue() + _IMM) < 512)) {
    _GETEOP1
        .setIntegerValue(_FIELD1[(_FIELD2.getIntegerValue() + _IMM)]
            .getIntegerValue());
  } else if (((_FIELD3.getIntegerValue() + _IMM) == 2048)) {
    _SOURCE_LEFT.request(_FIELD4);

    _FIELD4.addDoneListener(new IServiceDoneListener() {

      @Override
      public void done() {
        setActive(true);
        p1b();
      }

    });

    setActive(false);
```

```
    } else if (((_FIELD3.getIntegerValue() + _IMM) == 2049)) {
      _SOURCE_RIGHT.request(_FIELD10);

      _FIELD10.addDoneListener(new IServiceDoneListener() {

        @Override
        public void done() {
          setActive(true);
          p1b();
        }

      });

      setActive(false);
    } else if ((((_FIELD5.getIntegerValue() + _IMM) >= 2560) && ((_FIELD6
        .getIntegerValue() + _IMM) <= 2591))) {
      _FIELD8
          .setIntegerValue((int) (((_FIELD9.getIntegerValue() +
            _IMM) & 31)));
      _I2C.request(_FIELD7);
    }
  }
}

class SWI extends TYPE8 {
  protected final IServiceRequest _FIELD10;
  protected final IDataWord _FIELD9;
  protected final IDataWord _FIELD8;
  protected final IServiceRequest _FIELD7;
  protected final IDataWord _FIELD6;
  protected final IDataWord _FIELD5;
  protected final IServiceRequest _FIELD4;
  protected final IDataWord _FIELD3;
  protected final IDataWord _FIELD2;
  protected final IDataWord[] _FIELD1;
  protected final IDataWord _FIELD0;
  protected final IServiceModelInterface _I2C;
  protected final IServiceModelInterface _SINK_LEFT;
  protected final IServiceModelInterface _SINK_RIGHT;
  protected final int _IMM;
  protected final int _OP2;

  public SWI(IServiceModel p0, IService p1, int OP2, int OP1,
      int IMM, int EEXT1, int EXT1) {
    super(p0, p1, OP1, EEXT1, EXT1);
    _OP2 = OP2;
    _IMM = IMM;
    _SINK_LEFT = p0.getActiveServiceModelInterface("SINK_LEFT");
    _SINK_RIGHT = p0.getActiveServiceModelInterface("SINK_RIGHT");
    _I2C = p0.getActiveServiceModelInterface("I2C");
    _FIELD0 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP2).getElement());
    _FIELD1 = (IDataWord[]) fModel.getState().getMemorySegment(
        "mem").getElements();
    _FIELD2 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP2).getElement());
    _FIELD3 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP2).getElement());
    _FIELD4 = _SINK_LEFT.createServiceRequest("WRITE",
        new Object[] { _GETEOP1 });
    _FIELD5 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP2).getElement());
    _FIELD6 = ((IDataWord) fModel.getState().getRegister(
        "reg" + _OP2).getElement());
    _FIELD8 = new DataWord(24);
```

```
      _FIELD9 = ((IDataWord) fModel.getState().getRegister(
          "reg" + _OP2).getElement());
      _FIELD7 = _I2C.createServiceRequest("WRITE", new Object[] {
          _GETEOP1, _FIELD8 });
      _FIELD10 = _SINK_LEFT.createServiceRequest("WRITE",
          new Object[] { _GETEOP1 });
    }

    public final void p1() {
      if (((_FIELD0.getIntegerValue() + _IMM) < 512)) {
        _FIELD1[(_FIELD2.getIntegerValue() + _IMM)]
            .setIntegerValue(_GETEOP1.getIntegerValue());
      } else if (((_FIELD3.getIntegerValue() + _IMM) == 2304)) {
        _SINK_LEFT.request(_FIELD4);
      } else if (((_FIELD3.getIntegerValue() + _IMM) == 2305)) {
        _SINK_RIGHT.request(_FIELD10);
      } else if ((((_FIELD5.getIntegerValue() + _IMM) >= 2560) && ((_FIELD6
          .getIntegerValue() + _IMM) <= 2591))) {
        _FIELD8
            .setIntegerValue((int) (((_FIELD9.getIntegerValue() +
              _IMM) & 31)));
        _I2C.request(_FIELD7);
      }
    }
  }

  abstract class TYPE9 extends SVFServiceRequest {
    public TYPE9(IServiceModel p0, IService p1) {
      super(p0, p1);
    }

    public abstract void p1();
  }

  public IServiceRequest createServiceRequest(IService p0, Object p1[]) {
    if (p0.getIdentifier().equals("NOP"))
      return new NOP(fModel, p0);
    else if (p0.getIdentifier().equals("ADDS"))
      return new ADDS(fModel, p0, (Integer) p1[0], (Integer) p1[1],
          (Integer) p1[2], (Integer) p1[3], (Integer) p1[4]);
    else if (p0.getIdentifier().equals("ADDU"))
      return new ADDU(fModel, p0, (Integer) p1[0], (Integer) p1[1],
          (Integer) p1[2], (Integer) p1[3], (Integer) p1[4]);
    else if (p0.getIdentifier().equals("MULS"))
      return new MULS(fModel, p0, (Integer) p1[0], (Integer) p1[1],
          (Integer) p1[2], (Integer) p1[3], (Integer) p1[4]);
    else if (p0.getIdentifier().equals("MULU"))
      return new MULU(fModel, p0, (Integer) p1[0], (Integer) p1[1],
          (Integer) p1[2], (Integer) p1[3], (Integer) p1[4]);
    else if (p0.getIdentifier().equals("MACS"))
      return new MACS(fModel, p0, (Integer) p1[0], (Integer) p1[1],
          (Integer) p1[2], (Integer) p1[3], (Integer) p1[4]);
    else if (p0.getIdentifier().equals("MACU"))
      return new MACU(fModel, p0, (Integer) p1[0], (Integer) p1[1],
          (Integer) p1[2], (Integer) p1[3], (Integer) p1[4]);
    else if (p0.getIdentifier().equals("AND"))
      return new AND(fModel, p0, (Integer) p1[0], (Integer) p1[1],
          (Integer) p1[2], (Integer) p1[3], (Integer) p1[4]);
    else if (p0.getIdentifier().equals("OR"))
      return new OR(fModel, p0, (Integer) p1[0], (Integer) p1[1],
          (Integer) p1[2], (Integer) p1[3], (Integer) p1[4]);
    else if (p0.getIdentifier().equals("XOR"))
      return new XOR(fModel, p0, (Integer) p1[0], (Integer) p1[1],
          (Integer) p1[2], (Integer) p1[3], (Integer) p1[4]);
    else if (p0.getIdentifier().equals("LW"))
```

```
    return new LW(fModel, p0, (Integer) p1[0], (Integer) p1[1],
        (Integer) p1[2], (Integer) p1[3], (Integer) p1[4]);
  else if (p0.getIdentifier().equals("SW"))
    return new SW(fModel, p0, (Integer) p1[0], (Integer) p1[1],
        (Integer) p1[2], (Integer) p1[3], (Integer) p1[4]);
  else if (p0.getIdentifier().equals("ADDIS"))
    return new ADDIS(fModel, p0, (Integer) p1[0], (Integer) p1[1],
        (Integer) p1[2]);
  else if (p0.getIdentifier().equals("ADDIU"))
    return new ADDIU(fModel, p0, (Integer) p1[0], (Integer) p1[1],
        (Integer) p1[2]);
  else if (p0.getIdentifier().equals("MULIS"))
    return new MULIS(fModel, p0, (Integer) p1[0], (Integer) p1[1],
        (Integer) p1[2]);
  else if (p0.getIdentifier().equals("MULIU"))
    return new MULIU(fModel, p0, (Integer) p1[0], (Integer) p1[1],
        (Integer) p1[2]);
  else if (p0.getIdentifier().equals("MOVI"))
    return new MOVI(fModel, p0, (Integer) p1[0], (Integer) p1[1]);
  else if (p0.getIdentifier().equals("MOVUI"))
    return new MOVUI(fModel, p0, (Integer) p1[0], (Integer) p1[1]);
  else if (p0.getIdentifier().equals("LWA"))
    return new LWA(fModel, p0, (Integer) p1[0], (Integer) p1[1]);
  else if (p0.getIdentifier().equals("SWA"))
    return new SWA(fModel, p0, (Integer) p1[0], (Integer) p1[1]);
  else if (p0.getIdentifier().equals("ANDI"))
    return new ANDI(fModel, p0, (Integer) p1[0], (Integer) p1[1],
        (Integer) p1[2]);
  else if (p0.getIdentifier().equals("ORI"))
    return new ORI(fModel, p0, (Integer) p1[0], (Integer) p1[1],
        (Integer) p1[2]);
  else if (p0.getIdentifier().equals("XORI"))
    return new XORI(fModel, p0, (Integer) p1[0], (Integer) p1[1],
        (Integer) p1[2]);
  else if (p0.getIdentifier().equals("BGTZ"))
    return new BGTZ(fModel, p0, (Integer) p1[0], (Integer) p1[1]);
  else if (p0.getIdentifier().equals("BLEZ"))
    return new BLEZ(fModel, p0, (Integer) p1[0], (Integer) p1[1]);
  else if (p0.getIdentifier().equals("JMP"))
    return new JMP(fModel, p0, (Integer) p1[0]);
  else if (p0.getIdentifier().equals("SVF1"))
    return new SVF1(fModel, p0, (Integer) p1[0], (Integer) p1[1],
        (Integer) p1[2], (Integer) p1[3]);
  else if (p0.getIdentifier().equals("SVF2A"))
    return new SVF2A(fModel, p0, (Integer) p1[0], (Integer) p1[1],
        (Integer) p1[2], (Integer) p1[3]);
  else if (p0.getIdentifier().equals("SVF2B"))
    return new SVF2B(fModel, p0, (Integer) p1[0], (Integer) p1[1],
        (Integer) p1[2], (Integer) p1[3]);
  else if (p0.getIdentifier().equals("SLLI"))
    return new SLLI(fModel, p0, (Integer) p1[0], (Integer) p1[1],
        (Integer) p1[2], (Integer) p1[3]);
  else if (p0.getIdentifier().equals("SRLI"))
    return new SRLI(fModel, p0, (Integer) p1[0], (Integer) p1[1],
        (Integer) p1[2], (Integer) p1[3]);
  else if (p0.getIdentifier().equals("SRAI"))
    return new SRAI(fModel, p0, (Integer) p1[0], (Integer) p1[1],
        (Integer) p1[2], (Integer) p1[3]);
  else if (p0.getIdentifier().equals("BGT"))
    return new BGT(fModel, p0, (Integer) p1[0], (Integer) p1[1],
        (Integer) p1[2], (Integer) p1[3]);
  else if (p0.getIdentifier().equals("BLT"))
    return new BLT(fModel, p0, (Integer) p1[0], (Integer) p1[1],
        (Integer) p1[2], (Integer) p1[3]);
  else if (p0.getIdentifier().equals("BEQ"))
```

```
      return new BEQ(fModel, p0, (Integer) p1[0], (Integer) p1[1],
          (Integer) p1[2], (Integer) p1[3]);
    else if (p0.getIdentifier().equals("BNE"))
      return new BNE(fModel, p0, (Integer) p1[0], (Integer) p1[1],
          (Integer) p1[2], (Integer) p1[3]);
    else if (p0.getIdentifier().equals("MOV"))
      return new MOV(fModel, p0, (Integer) p1[0], (Integer) p1[1],
          (Integer) p1[2], (Integer) p1[3], (Integer) p1[4],
          (Integer) p1[5]);
    else if (p0.getIdentifier().equals("ABS"))
      return new ABS(fModel, p0, (Integer) p1[0], (Integer) p1[1],
          (Integer) p1[2], (Integer) p1[3], (Integer) p1[4],
          (Integer) p1[5]);
    else if (p0.getIdentifier().equals("LWI"))
      return new LWI(fModel, p0, (Integer) p1[0], (Integer) p1[1],
          (Integer) p1[2], (Integer) p1[3], (Integer) p1[4]);
    else if (p0.getIdentifier().equals("SWI"))
      return new SWI(fModel, p0, (Integer) p1[0], (Integer) p1[1],
          (Integer) p1[2], (Integer) p1[3], (Integer) p1[4]);
    return null;
  }
}

class SoftwareInterface extends AbstractServiceModelInterface {
  public SoftwareInterface() {
    super("SWINT");
    this.addService(new Service("NOP"));
    this.addService(new Service("ADDS"));
    this.addService(new Service("ADDU"));
    this.addService(new Service("MULS"));
    this.addService(new Service("MULU"));
    this.addService(new Service("MACS"));
    this.addService(new Service("MACU"));
    this.addService(new Service("AND"));
    this.addService(new Service("OR"));
    this.addService(new Service("XOR"));
    this.addService(new Service("LW"));
    this.addService(new Service("SW"));
    this.addService(new Service("ADDIS"));
    this.addService(new Service("ADDIU"));
    this.addService(new Service("MULIS"));
    this.addService(new Service("MULIU"));
    this.addService(new Service("MOVI"));
    this.addService(new Service("MOVUI"));
    this.addService(new Service("LWA"));
    this.addService(new Service("SWA"));
    this.addService(new Service("ANDI"));
    this.addService(new Service("ORI"));
    this.addService(new Service("XORI"));
    this.addService(new Service("BGTZ"));
    this.addService(new Service("BLEZ"));
    this.addService(new Service("JMP"));
    this.addService(new Service("SVF1"));
    this.addService(new Service("SVF2A"));
    this.addService(new Service("SVF2B"));
    this.addService(new Service("SLLI"));
    this.addService(new Service("SRLI"));
    this.addService(new Service("SRAI"));
    this.addService(new Service("BGT"));
    this.addService(new Service("BLT"));
    this.addService(new Service("BEQ"));
    this.addService(new Service("BNE"));
    this.addService(new Service("MOV"));
    this.addService(new Service("ABS"));
    this.addService(new Service("LWI"));
```

```
    this.addService(new Service("SWI"));
  }

  public IServiceRequest createServiceRequest(long inst) {
    if ((inst & 0) == 0 && (~inst & 16777215) == 16777215) {
      return this.createServiceRequest("NOP", new Object[] {});
    } else if ((inst & 262144) == 262144
        && (~inst & 16252928) == 16252928) {
      int l0 = (int) (inst & 960) >>> 6;
      int l1 = (int) (inst & 15360) >>> 10;
      int l2 = (int) (inst & 245760) >>> 14;
      int l3 = (int) (inst & 56) >>> 3;
      int l4 = (int) (inst & 7) >>> 0;
      return this.createServiceRequest("ADDS", new Object[] { l0, l1,
          l2, l3, l4 });
    } else if ((inst & 524288) == 524288
        && (~inst & 15990784) == 15990784) {
      int l5 = (int) (inst & 960) >>> 6;
      int l6 = (int) (inst & 15360) >>> 10;
      int l7 = (int) (inst & 245760) >>> 14;
      int l8 = (int) (inst & 56) >>> 3;
      int l9 = (int) (inst & 7) >>> 0;
      return this.createServiceRequest("ADDU", new Object[] { l5, l6,
          l7, l8, l9 });
    } else if ((inst & 786432) == 786432
        && (~inst & 15728640) == 15728640) {
      int l10 = (int) (inst & 960) >>> 6;
      int l11 = (int) (inst & 15360) >>> 10;
      int l12 = (int) (inst & 245760) >>> 14;
      int l13 = (int) (inst & 56) >>> 3;
      int l14 = (int) (inst & 7) >>> 0;
      return this.createServiceRequest("MULS", new Object[] { l10,
          l11, l12, l13, l14 });
    } else if ((inst & 1048576) == 1048576
        && (~inst & 15466496) == 15466496) {
      int l15 = (int) (inst & 960) >>> 6;
      int l16 = (int) (inst & 15360) >>> 10;
      int l17 = (int) (inst & 245760) >>> 14;
      int l18 = (int) (inst & 56) >>> 3;
      int l19 = (int) (inst & 7) >>> 0;
      return this.createServiceRequest("MULU", new Object[] { l15,
          l16, l17, l18, l19 });
    } else if ((inst & 1310720) == 1310720
        && (~inst & 15204352) == 15204352) {
      int l20 = (int) (inst & 960) >>> 6;
      int l21 = (int) (inst & 15360) >>> 10;
      int l22 = (int) (inst & 245760) >>> 14;
      int l23 = (int) (inst & 56) >>> 3;
      int l24 = (int) (inst & 7) >>> 0;
      return this.createServiceRequest("MACS", new Object[] { l20,
          l21, l22, l23, l24 });
    } else if ((inst & 1572864) == 1572864
        && (~inst & 14942208) == 14942208) {
      int l25 = (int) (inst & 960) >>> 6;
      int l26 = (int) (inst & 15360) >>> 10;
      int l27 = (int) (inst & 245760) >>> 14;
      int l28 = (int) (inst & 56) >>> 3;
      int l29 = (int) (inst & 7) >>> 0;
      return this.createServiceRequest("MACU", new Object[] { l25,
          l26, l27, l28, l29 });
    } else if ((inst & 9175040) == 9175040
        && (~inst & 7340032) == 7340032) {
      int l30 = (int) (inst & 960) >>> 6;
      int l31 = (int) (inst & 15360) >>> 10;
      int l32 = (int) (inst & 245760) >>> 14;
```

```
    int l33 = (int) (inst & 56) >>> 3;
    int l34 = (int) (inst & 7) >>> 0;
    return this.createServiceRequest("AND", new Object[] { l30,
        l31, l32, l33, l34 });
  } else if ((inst & 9437184) == 9437184
      && (~inst & 7077888) == 7077888) {
    int l35 = (int) (inst & 960) >>> 6;
    int l36 = (int) (inst & 15360) >>> 10;
    int l37 = (int) (inst & 245760) >>> 14;
    int l38 = (int) (inst & 56) >>> 3;
    int l39 = (int) (inst & 7) >>> 0;
    return this.createServiceRequest("OR", new Object[] { l35, l36,
        l37, l38, l39 });
  } else if ((inst & 9699328) == 9699328
      && (~inst & 6815744) == 6815744) {
    int l40 = (int) (inst & 960) >>> 6;
    int l41 = (int) (inst & 15360) >>> 10;
    int l42 = (int) (inst & 245760) >>> 14;
    int l43 = (int) (inst & 56) >>> 3;
    int l44 = (int) (inst & 7) >>> 0;
    return this.createServiceRequest("XOR", new Object[] { l40,
        l41, l42, l43, l44 });
  } else if ((inst & 6291456) == 6291456
      && (~inst & 10223616) == 10223616) {
    int l45 = (int) (inst & 960) >>> 6;
    int l46 = (int) (inst & 15360) >>> 10;
    int l47 = (int) (inst & 245760) >>> 14;
    int l48 = (int) (inst & 56) >>> 3;
    int l49 = (int) (inst & 7) >>> 0;
    return this.createServiceRequest("LW", new Object[] { l45, l46,
        l47, l48, l49 });
  } else if ((inst & 6029312) == 6029312
      && (~inst & 10485760) == 10485760) {
    int l50 = (int) (inst & 960) >>> 6;
    int l51 = (int) (inst & 15360) >>> 10;
    int l52 = (int) (inst & 245760) >>> 14;
    int l53 = (int) (inst & 56) >>> 3;
    int l54 = (int) (inst & 7) >>> 0;
    return this.createServiceRequest("SW", new Object[] { l50, l51,
        l52, l53, l54 });
  } else if ((inst & 1835008) == 1835008
      && (~inst & 14692352) == 14692352) {
    int l55 = (int) (inst & 15360) >>> 10;
    int l56 = (int) (inst & 245760) >>> 14;
    int l57 = (int) (inst & 4095) >>> 0;
    return this.createServiceRequest("ADDIS", new Object[] { l55,
        l56, l57 });
  } else if ((inst & 2097152) == 2097152
      && (~inst & 14430208) == 14430208) {
    int l58 = (int) (inst & 15360) >>> 10;
    int l59 = (int) (inst & 245760) >>> 14;
    int l60 = (int) (inst & 4095) >>> 0;
    return this.createServiceRequest("ADDIU", new Object[] { l58,
        l59, l60 });
  } else if ((inst & 2359296) == 2359296
      && (~inst & 14168064) == 14168064) {
    int l61 = (int) (inst & 15360) >>> 10;
    int l62 = (int) (inst & 245760) >>> 14;
    int l63 = (int) (inst & 4095) >>> 0;
    return this.createServiceRequest("MULIS", new Object[] { l61,
        l62, l63 });
  } else if ((inst & 2621440) == 2621440
      && (~inst & 13905920) == 13905920) {
    int l64 = (int) (inst & 15360) >>> 10;
    int l65 = (int) (inst & 245760) >>> 14;
```

```
    int l66 = (int) (inst & 4095) >>> 0;
    return this.createServiceRequest("MULIU", new Object[] { l64,
        l65, l66 });
  } else if ((inst & 3932160) == 3932160
      && (~inst & 12595200) == 12595200) {
    int l67 = (int) (inst & 245760) >>> 14;
    int l68 = (int) (inst & 4095) >>> 0;
    return this.createServiceRequest("MOVI", new Object[] { l67,
        l68 });
  } else if ((inst & 4194304) == 4194304
      && (~inst & 12333056) == 12333056) {
    int l69 = (int) (inst & 245760) >>> 14;
    int l70 = (int) (inst & 4095) >>> 0;
    return this.createServiceRequest("MOVUI", new Object[] { l69,
        l70 });
  } else if ((inst & 7340032) == 7340032
      && (~inst & 9187328) == 9187328) {
    int l71 = (int) (inst & 4095) >>> 0;
    int l72 = (int) (inst & 245760) >>> 14;
    return this.createServiceRequest("LWA",
        new Object[] { l71, l72 });
  } else if ((inst & 7077888) == 7077888
      && (~inst & 9449472) == 9449472) {
    int l73 = (int) (inst & 4095) >>> 0;
    int l74 = (int) (inst & 245760) >>> 14;
    return this.createServiceRequest("SWA",
        new Object[] { l73, l74 });
  } else if ((inst & 9961472) == 9961472
      && (~inst & 6565888) == 6565888) {
    int l75 = (int) (inst & 15360) >>> 10;
    int l76 = (int) (inst & 245760) >>> 14;
    int l77 = (int) (inst & 4095) >>> 0;
    return this.createServiceRequest("ANDI", new Object[] { l75,
        l76, l77 });
  } else if ((inst & 10223616) == 10223616
      && (~inst & 6303744) == 6303744) {
    int l78 = (int) (inst & 15360) >>> 10;
    int l79 = (int) (inst & 245760) >>> 14;
    int l80 = (int) (inst & 4095) >>> 0;
    return this.createServiceRequest("ORI", new Object[] { l78,
        l79, l80 });
  } else if ((inst & 10485760) == 10485760
      && (~inst & 6041600) == 6041600) {
    int l81 = (int) (inst & 15360) >>> 10;
    int l82 = (int) (inst & 245760) >>> 14;
    int l83 = (int) (inst & 4095) >>> 0;
    return this.createServiceRequest("XORI", new Object[] { l81,
        l82, l83 });
  } else if ((inst & 5242880) == 5242880
      && (~inst & 11284480) == 11284480) {
    int l84 = (int) (inst & 245760) >>> 14;
    int l85 = (int) (inst & 4095) >>> 0;
    return this.createServiceRequest("BGTZ", new Object[] { l84,
        l85 });
  } else if ((inst & 5505024) == 5505024
      && (~inst & 11022336) == 11022336) {
    int l86 = (int) (inst & 245760) >>> 14;
    int l87 = (int) (inst & 4095) >>> 0;
    return this.createServiceRequest("BLEZ", new Object[] { l86,
        l87 });
  } else if ((inst & 5767168) == 5767168
      && (~inst & 10747904) == 10747904) {
    int l88 = (int) (inst & 262143) >>> 0;
    return this.createServiceRequest("JMP", new Object[] { l88 });
  } else if ((inst & 4456448) == 4456448
```

```
    && (~inst & 12059584) == 12059584) {
  int l89 = (int) (inst & 15360) >>> 10;
  int l90 = (int) (inst & 245760) >>> 14;
  int l91 = (int) (inst & 56) >>> 3;
  int l92 = (int) (inst & 7) >>> 0;
  return this.createServiceRequest("SVF1", new Object[] { l89,
      l90, l91, l92 });
} else if ((inst & 4718592) == 4718592
    && (~inst & 11797440) == 11797440) {
  int l93 = (int) (inst & 15360) >>> 10;
  int l94 = (int) (inst & 245760) >>> 14;
  int l95 = (int) (inst & 56) >>> 3;
  int l96 = (int) (inst & 7) >>> 0;
  return this.createServiceRequest("SVF2A", new Object[] { l93,
      l94, l95, l96 });
} else if ((inst & 4980736) == 4980736
    && (~inst & 11535296) == 11535296) {
  int l97 = (int) (inst & 15360) >>> 10;
  int l98 = (int) (inst & 245760) >>> 14;
  int l99 = (int) (inst & 56) >>> 3;
  int l100 = (int) (inst & 7) >>> 0;
  return this.createServiceRequest("SVF2B", new Object[] { l97,
      l98, l99, l100 });
} else if ((inst & 2883584) == 2883584
    && (~inst & 13631488) == 13631488) {
  int l101 = (int) (inst & 15360) >>> 10;
  int l102 = (int) (inst & 245760) >>> 14;
  int l103 = (int) (inst & 1016) >>> 3;
  int l104 = (int) (inst & 7) >>> 0;
  return this.createServiceRequest("SLLI", new Object[] { l101,
      l102, l103, l104 });
} else if ((inst & 3145728) == 3145728
    && (~inst & 13369344) == 13369344) {
  int l105 = (int) (inst & 15360) >>> 10;
  int l106 = (int) (inst & 245760) >>> 14;
  int l107 = (int) (inst & 1016) >>> 3;
  int l108 = (int) (inst & 7) >>> 0;
  return this.createServiceRequest("SRLI", new Object[] { l105,
      l106, l107, l108 });
} else if ((inst & 3407872) == 3407872
    && (~inst & 13107200) == 13107200) {
  int l109 = (int) (inst & 15360) >>> 10;
  int l110 = (int) (inst & 245760) >>> 14;
  int l111 = (int) (inst & 1016) >>> 3;
  int l112 = (int) (inst & 7) >>> 0;
  return this.createServiceRequest("SRAI", new Object[] { l109,
      l110, l111, l112 });
} else if ((inst & 8126464) == 8126464
    && (~inst & 8388608) == 8388608) {
  int l113 = (int) (inst & 15360) >>> 10;
  int l114 = (int) (inst & 245760) >>> 14;
  int l115 = (int) (inst & 1016) >>> 3;
  int l116 = (int) (inst & 7) >>> 0;
  return this.createServiceRequest("BGT", new Object[] { l113,
      l114, l115, l116 });
} else if ((inst & 8388608) == 8388608
    && (~inst & 8126464) == 8126464) {
  int l117 = (int) (inst & 15360) >>> 10;
  int l118 = (int) (inst & 245760) >>> 14;
  int l119 = (int) (inst & 1016) >>> 3;
  int l120 = (int) (inst & 7) >>> 0;
  return this.createServiceRequest("BLT", new Object[] { l117,
      l118, l119, l120 });
} else if ((inst & 8650752) == 8650752
    && (~inst & 7864320) == 7864320) {
```

```
      int l121 = (int) (inst & 15360) >>> 10;
      int l122 = (int) (inst & 245760) >>> 14;
      int l123 = (int) (inst & 1016) >>> 3;
      int l124 = (int) (inst & 7) >>> 0;
      return this.createServiceRequest("BEQ", new Object[] { l121,
          l122, l123, l124 });
    } else if ((inst & 8650752) == 8650752
      && (~inst & 7864320) == 7864320) {
      int l125 = (int) (inst & 15360) >>> 10;
      int l126 = (int) (inst & 245760) >>> 14;
      int l127 = (int) (inst & 1016) >>> 3;
      int l128 = (int) (inst & 7) >>> 0;
      return this.createServiceRequest("BNE", new Object[] { l125,
          l126, l127, l128 });
    } else if ((inst & 3670016) == 3670016
      && (~inst & 12845248) == 12845248) {
      int l129 = (int) (inst & 15360) >>> 10;
      int l130 = (int) (inst & 245760) >>> 14;
      int l131 = (int) (inst & 512) >>> 9;
      int l132 = (int) (inst & 256) >>> 8;
      int l133 = (int) (inst & 56) >>> 3;
      int l134 = (int) (inst & 7) >>> 0;
      return this.createServiceRequest("MOV", new Object[] { l129,
          l130, l131, l132, l133, l134 });
    } else if ((inst & 7602176) == 7602176
      && (~inst & 8913088) == 8913088) {
      int l135 = (int) (inst & 15360) >>> 10;
      int l136 = (int) (inst & 245760) >>> 14;
      int l137 = (int) (inst & 512) >>> 9;
      int l138 = (int) (inst & 256) >>> 8;
      int l139 = (int) (inst & 56) >>> 3;
      int l140 = (int) (inst & 7) >>> 0;
      return this.createServiceRequest("ABS", new Object[] { l135,
          l136, l137, l138, l139, l140 });
    } else if ((inst & 6815744) == 6815744
      && (~inst & 9699840) == 9699840) {
      int l141 = (int) (inst & 15360) >>> 10;
      int l142 = (int) (inst & 245760) >>> 14;
      int l143 = (int) (inst & 248) >>> 3;
      int l144 = (int) (inst & 256) >>> 8;
      int l145 = (int) (inst & 7) >>> 0;
      return this.createServiceRequest("LWI", new Object[] { l141,
          l142, l143, l144, l145 });
    } else if ((inst & 6553600) == 6553600
      && (~inst & 9961984) == 9961984) {
      int l146 = (int) (inst & 15360) >>> 10;
      int l147 = (int) (inst & 245760) >>> 14;
      int l148 = (int) (inst & 248) >>> 3;
      int l149 = (int) (inst & 256) >>> 8;
      int l150 = (int) (inst & 7) >>> 0;
      return this.createServiceRequest("SWI", new Object[] { l146,
          l147, l148, l149, l150 });
    }
    return null;
  }
}

public class SVF3ServiceModelState extends AbstractServiceModelState {
  public SVF3ServiceModelState() {
    IRegisterGroup general = StateFactory
        .createRegisterGroup("general");
    fGroups.add(general);
    IRegisterGroup b_hp = StateFactory.createRegisterGroup("b_hp");
    fGroups.add(b_hp);
    for (int i = 0; i < 16; i++) {
```

```
  b_hp.addMember(StateFactory.createRegister("b_hp" + i,
      new DataWord(0, 24)));
}
IRegisterGroup cfgst = StateFactory.createRegisterGroup("cfgst");
fGroups.add(cfgst);
for (int i = 0; i < 8; i++) {
  cfgst.addMember(StateFactory.createRegister("cfgst" + i,
      new DataWord(0, 24)));
}
IRegisterGroup b_bp = StateFactory.createRegisterGroup("b_bp");
fGroups.add(b_bp);
for (int i = 0; i < 16; i++) {
  b_bp.addMember(StateFactory.createRegister("b_bp" + i,
      new DataWord(0, 24)));
}
IRegisterGroup reg = StateFactory.createRegisterGroup("reg");
fGroups.add(reg);
for (int i = 0; i < 16; i++) {
  reg.addMember(StateFactory.createRegister("reg" + i,
      new DataWord(0, 24)));
}
IRegisterGroup a_hp = StateFactory.createRegisterGroup("a_hp");
fGroups.add(a_hp);
for (int i = 0; i < 16; i++) {
  a_hp.addMember(StateFactory.createRegister("a_hp" + i,
      new DataWord(0, 24)));
}
IRegisterGroup c2 = StateFactory.createRegisterGroup("c2");
fGroups.add(c2);
for (int i = 0; i < 16; i++) {
  c2.addMember(StateFactory.createRegister("c2" + i,
      new DataWord(0, 24)));
}
IRegisterGroup a_bp = StateFactory.createRegisterGroup("a_bp");
fGroups.add(a_bp);
for (int i = 0; i < 16; i++) {
  a_bp.addMember(StateFactory.createRegister("a_bp" + i,
      new DataWord(0, 24)));
}
IRegisterGroup c1 = StateFactory.createRegisterGroup("c1");
fGroups.add(c1);
for (int i = 0; i < 16; i++) {
  c1.addMember(StateFactory.createRegister("c1" + i,
      new DataWord(0, 24)));
}
IRegisterGroup b_lp = StateFactory.createRegisterGroup("b_lp");
fGroups.add(b_lp);
for (int i = 0; i < 16; i++) {
  b_lp.addMember(StateFactory.createRegister("b_lp" + i,
      new DataWord(0, 24)));
}
IRegister pc = StateFactory.createRegister("pc",
    new DataWord(0, 16));
general.addMember(pc);

IDataWord mem[] = new DataWord[512];

for (int i = 0; i < 512; i++) {
  mem[i] = new DataWord(0, 24);
}
fMemories.add(StateFactory.createMemory("mem", mem, 0, 511));

IRegisterGroup a_lp = StateFactory.createRegisterGroup("a_lp");
fGroups.add(a_lp);
for (int i = 0; i < 16; i++) {
```

```
        a_lp.addMember(StateFactory.createRegister("a_lp" + i,
            new DataWord(0, 24)));
      }
      IRegisterGroup fcfg = StateFactory.createRegisterGroup("fcfg");
      fGroups.add(fcfg);
      for (int i = 0; i < 16; i++) {
        fcfg.addMember(StateFactory.createRegister("fcfg" + i,
            new DataWord(0, 24)));
      }
    }
  }
}
```

Bibliography

[1] Gcc, the gnu compiler collection.
`http://gcc.gnu.org/`.

[2] Matlab.
`http://www.matlab.com`.

[3] Synopsys virtual prototyping.
`http://www.synopsys.com/Tools/SLD/VirtualPrototyping`.

[4] Target.
`http://www.retarget.com/`.

[5] Tensilica.
`http://www.tensilica.com/`.

[6] Wind river - simics.
`http://www.windriver.com/products/simics/`.

[7] F. Angiolini, J. Ceng, R. Leupers, F. Ferrari, C. Ferri, and L. Benini. An integrated open framework for heterogeneous mpsoc design space exploration. *Proceedings of the Design Automation & Test in Europe Conference*, 1:1–6, 2006.

[8] GNU ARM. Gnu arm toolchain, 2009. `http://www.gnuarm.com/`.

[9] Semiconductor Industry Association. International technology roadmap for semiconductors, 1999. `http://www.itrs.net`.

[10] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. The archc architecture description language and tools. *International Journal of Parallel Programming*, 33:453–484, 2005. 10.1007/s10766-005-7301-0.

[11] A. Bakshi, V. K. Prasanna, and A. Ledeczi. Milan: A model based integrated simulation framework for design of embedded systems. *SIGPLAN Not.*, 36(8):82–93, 2001.

[12] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara. *Hardware-software co-design of embedded systems: the POLIS approach.* Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[13] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *Computer*, 36(4):45–52+4, 2003.

[14] Luca Benini, Davide Bertozzi, Alessandro Bogliolo, Francesco Menichelli, and Mauro Olivieri. Mparm: Exploring the multi-processor soc design space with systemc. *The Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, 41(2):169–182, 2005.

[15] J.-Y. Brunel, W.M. Kruijtzer, H.J.H.N. Kenter, F. Petrot, L. Pasquier, E.A. de Kock, and W.J.M. Smits. Cosy communication ip's. pages 406 –409, 2000.

[16] L. Cai and D. Gajski. Transaction level modeling: an overview. pages 19 – 24, oct. 2003.

[17] Christos G. Cassandras. *Discrete event systems: Modeling and performance analysis.* Aksen Associates, 1993.

[18] A.S. Cassidy, J.M. Paul, and D.E. Thomas. Layered, multi-threaded, high-level performance design. *2003 Design, Automation and Test in Europe Conference and Exhibition*, pages 954–959, 2003.

[19] Jianjiang Ceng, Weihua Sheng, Jeronimo Castrillon, Anastasia Stulova, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. A high-level virtual platform for early mpsoc software development. In *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 11–20, New York, NY, USA, 2009. ACM.

[20] W.O. Cesario, D. Lyonnard, G. Nicolescu, Y. Paviot, Sungjoo Yoo, A.A. Jerraya, L. Gauthier, and M. Diaz-Nava. Multiprocessor soc platforms: a component-based design approach. *IEEE Design and Test of Computers*, 19(6):52–63, 2002.

[21] Carlos A. Coello Coello. A comprehensive survey of evolutionary-based multiobjective optimization techniques. *Knowledge and Information Systems*, 1:269–308, 1998.

[22] Joseph E. Coffland and Andy D. Pimentel. A software framework for efficient system-level performance evaluation of embedded systems. *Proceedings of the ACM Symposium on Applied Computing*, pages 666–671, 2003.

[23] Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, Marc Massot, Sandra Moral, Claudio Passerone, Yosinori Watanabe, and Alberto Sangiovanni-Vincentelli. Task generation and compile-time scheduling for mixed data-control embedded software. In *DAC '00: Proceedings of the 37th Annual Design Automation Conference*, pages 489–494, New York, NY, USA, 2000. ACM.

[24] E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, and K.A. Vissers. Yapi: application modeling for signal processing systems. *Proceedings - Design Automation Conference*, pages 402–405, 2000.

[25] Kalyanmoy Deb. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, chapter Multi-Objective Optimization, Chapter 10, pages 97–1–25. Springer, 2005.

[26] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast elitist multi-objective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6:182–197, 2000.

[27] D. Densmore, A. Simalatsar, A. Davare, R. Passerone, and A. Sangiovanni-Vincentelli. Umts mpsoc design evaluation using a system level design framework. *2009 Design, Automation Test in Europe Conference Exhibition*, pages 478–483, 2009.

[28] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[29] Joachim Falk, Christian Haubelt, and Jürgen Teich. Efficient representation and simulation of model-based designs in systemc. *In Proc. FDL 06a*, pages 129–134, 2006.

[30] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nml. *European Design and Test Conference, 1995. EDTC 1995, Proceedings.*, pages 503–507, 1995.

[31] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nml. In *Proceedings of the 1995 European conference on Design and Test*, EDTC '95, pages 503–, Washington, DC, USA, 1995. IEEE Computer Society.

[32] Eclipse Foundation. Eclipse. `http://www.eclipse.org/`.

[33] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology.* Kluwer Academic Publishers, January 2000.

[34] Daniel Gajski, Jianwen Zhu, and Rainer Dömer. Essential issues in codesign. Technical report, University of California, Irvine, 1997.

[35] D.D. Gajski and R.H. Kuhn. New vlsi tools. *Computer*, 16(12):11 –14, dec. 1983.

[36] Lei Gao, K. Karuri, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. Multiprocessor performance estimation using hybrid simulation. *2008 45th ACM/IEEE Design Automation Conference*, pages 325–330, 2008.

[37] T. Givargis and F. Vahid. Platune: a tuning framework for system-on-a-chip platforms. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 21(11):1317 – 1327, November 2002.

[38] J. Grode and J. Madsen. A unified component modeling approach for performance estimation in hardware/software codesign. volume 1, pages 65–69. IEEE, 1998.

[39] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. Isdl: an instruction set description language for retargetability. In *Proceedings of the 34th annual Design Automation Conference*, DAC '97, pages 299–302, New York, NY, USA, 1997. ACM.

[40] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. Expression: a language for architecture exploration through compiler/simulator retargetability. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '99, New York, NY, USA, 1999. ACM.

[41] M.R. Hartoog, J.A. Rowson, P.D. Reddy, S. Desai, D.D. Dunlop, E.A. Harcourt, and N. Khullar. Generation of software tools from processor descriptions for hardware/software codesign. In *Design Automation Conference, 1997. Proceedings of the 34th*, pages 303 –306, June 1997.

[42] INTEL. Intel xscale core - developer's manual, 2004. `http://www.intel.com`.

[43] Axel Jantsch. *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation.* Systems on Silicon. Morgan Kaufmann, 2003.

[44] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts.* Springer-Verlag, 1992.

[45] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[46] Tero Kangas, Petri Kukkala, Heikki Orsila, Erno Salminen, Marko Hännikäinen, Timo D. Hämäläinen, Jouni Riihimäki, and Kimmo Kuusilinna. Uml-based multiprocessor soc design framework. *ACM Trans. Embed. Comput. Syst.*, 5(2):281–320, 2006.

[47] Joachim Keinert, Martin Strübar, Thomas Schlichter, Joachim Falk, Jens Gladigau, Christian Haubelt, Jürgen Teich, and Michael Meredith. Systemcodesigner—an automatic esl synthesis approach by design space exploration and behavioral synthesis for streaming applications. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):1–23, 2009.

[48] K. Keutzer, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, 2000.

[49] V. Kianzad and S.S. Bhattacharyya. Charmed: a multi-objective co-synthesis framework for multi-mode embedded systems. pages 28 – 40, sep. 2004.

[50] B. Kienhuis, E. Deprettere, K. Vissers, and P. Van Der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. *Proceedings IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 338–349, 1997.

[51] B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: deriving process networks from matlab for embedded signal processing architectures. pages 13–17, 2000.

[52] Tim Kogel, Malte Doerper, Torsten Kempf, Andreas Wieferink, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Samos iii – architectures and implementation - virtual architecture mapping: A systemc based methodology for architectural exploration of system-on-chip designs. *Lecture Notes in Computer Science*, 3133:138, 2004.

[53] Tim Kogel, Andreas Wieferink, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Virtual architecture mapping: A systemc based methodology for architectural exploration of system-on-chip designs. In *in Proc. of the Int. workshop on Systems, Architectures, Modeling and Simulation (SAMOS*, pages 138–148, 2003.

[54] Abdullah Konak, David W. Coit, and Alice E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering & System Safety*, 91(9):992–1007, September 2006.

[55] Daniel Kästner. Tdl: A hardware description language for retargetable postpass optimizations and analyses. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering*, volume 2830 of *Lecture Notes in Computer Science*, pages 18–36. Springer Berlin / Heidelberg, 2003.

[56] S. Kunzli, F. Poletti, L. Benini, and L. Thiele. Combining simulation and formal methods for system-level performance analysis. *Proceedings of the Design Automation & Test in Europe Conference*, 1:1–6, 2006.

[57] K. Lahiri, A. Raghunathan, and S. Dey. System-level performance analysis for designing on-chip communication architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(6):768–783, 2001.

[58] Kanishka Lahiri, Anand Raghunathan, and Sujit Dey. Performance analysis of systems with multi-channel communication architectures. In *in Proc. Int. Conf. VLSI Design*, pages 530–537, 2000.

[59] E.A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(12):1217 –1229, dec. 1998.

[60] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, 1987.

[61] Rainer Leupers and Peter Marwedel. Retargetable code generation based on structural processor description. *Design Automation for Embedded Systems*, 3:75–108, 1998. 10.1023/A:1008807631619.

[62] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Deprettere. System level design with spade: an m-jpeg case study. *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281)*, pages 31–38, 2001.

[63] P. Lieverse, P. Van Der Wolf, E. Deprettere, and K. Vissers. A methodology for architecture exploration of heterogeneous signal processing systems. *1999 IEEE Workshop on Signal Processing Systems. SiPS 99. Design and Implementation (Cat. No.99TH8461)*, pages 181–190, 1999.

[64] P. Lieverse, P. Van Der Wolf, E. Deprettere, and K. Vissers. A methodology for architecture exploration of heterogeneous signal processing systems. *The Journal of VLSI Signal Processing*, 29(3):197–207, 2001.

[65] ARM Co. Ltd. Arm7tdmi technical reference manual, 2004. `http://www.arm.com`.

[66] ARM Co. Ltd. Arm9tdmi technical reference manual, 2005. `http://www.arm.com`.

[67] ARM Co. Ltd. Arm architecture reference manual, 2006. `http://www.arm.com`.

[68] Shankar Mahadevan, Kashif Virk, and Jan Madsen. Arts: A systemc-based framework for multiprocessor systems-on-chip modelling. *Design Automation for Embedded Systems*, 11(4):285–311, 2007.

[69] G. Martin. Design methodologies for system level ip. *Proceedings Design, Automation and Test in Europe*, pages 286–289, 1998.

[70] G. Martin. Overview of the mpsoc design challenge. *2006 43rd ACM/IEEE Design Automation Conference*, pages 274–279, 2006.

[71] Giovanni De Micheli and Rajesh K. Gupta. Hardware/software co-design. In *Proceedings of the IEEE*, volume 85, pages 349–365, March 1997.

[72] A. Mihal, C. Kulkarni, M. Moskewicz, M. Tsai, N. Shah, S. Weber, Yujia Jin, K. Keutzer, K. Vissers, C. Sauer, and S. Malik. Developing architectural platforms: a disciplined approach. *IEEE Design and Test of Computers*, 19(6):6–16, 2002.

[73] Sun Mircrosystems. Java se 1.6, 2008. `http://www.java.sun.com`.

[74] Marius Monton, Antoni Portero, Marc Moreno, Borja Martinez, and Jordi Carrabina. Mixed sw/systemc soc emulation framework. pages 2338 – 2341, jun. 2007.

[75] Gordon E. Moore. Craming more components onto integraed circuits. *Electronics*, 38, April 1965.

[76] Hendrik Lambertus Muller. *Simulating Computer Architectures.* PhD thesis, Dept. of Computer Science, Univ. of Amsterdam, 1993.

[77] H. Nikolov, T. Stefanov, and E. Deprettere. Multi-processor system design with espam. *2006 4th IEEE/ACM/IFIPHardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 211–216, 2007.

[78] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):542–555, 2008.

[79] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: Toward composable multimedia mp-soc design. *2008 45th ACM/IEEE Design Automation Conference*, pages 574–579, 2008.

[80] Open SystemC Initiative OSCI. Transaction-level modeling standard 2.0, 2009. http://www.systemc.org/downloads/standards/tlm20/.

[81] Open SystemC Initiative OSCI. Osci home page, 2010. http://www.systemc.org.

[82] M. Oyamada, F.R. Wagner, M. Bonaciu, W. Cesario, and A. Jerraya. Software performance estimation in mpsoc design. pages 38 –43, jan. 2007.

[83] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. Multi-objective design space exploration of embedded systems. *J. Embedded Comput.*, 1:305–316, August 2005.

[84] J. Peng, S. Abdi, and D. Gajski. Automatic model refinement for fast architecture exploration [soc design]. *Proceedings of ASP-DAC/VLSI Design 2002. 7th Asia and South Pacific Design Automation Conference and 15h International Conference on VLSI Design*, pages 332–337, 2002.

[85] A D Pimentel, S Polstra, F Terpstra, A W van Halderen, J E Coffland, and L O Hertzberger. A system-level design and simulation - towards efficient design space exploration of heterogeneous embedded media systems. *Lecture Notes in Computer Science*, 2268:57, 2002.

[86] A.D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*, 55(2):99–112, 2006.

[87] A.D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*, 55(2):99–112, 2006.

[88] A.D. Pimentel, L.O. Hertzbetger, P. Lieverse, P. van der Wolf, and E.E. Deprettere. Exploring embedded-systems architectures with artemis. *Computer*, 34(11):57–63, 2001.

[89] Simon Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Comput.*, 55(2):99–112, 2006. Member-Pimentel, Andy D. and Student Member-Erbas, Cagkan.

[90] L. Biro B. Bowhill E. Fetzer P. Gronowski T. Grutkowski R. J. Riedlinger, R. Bhatia. A 32nm 3.1 billion transistor 12-wide-issue itanium processor. *2011 IEEE International Solid-State circuits conference*, 2011.

[91] Victor Reyes, Tomas Bautista, Gustavo Marrero, Pedro P. Carballo, and Wido Kruijtzer. Casse: A system-level modeling and design-space exploration tool for multiprocessor systems-on-chip. *Proceedings of the EU-*

ROMICRO Systems on Digital System Design, DSD 2004, pages 476–483, 2004.

[92] A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *IEEE Design and Test of Computers*, 18(6):23–33, 2001.

[93] K. Sastry, D. Goldberg, and G. Kendall. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, chapter Genetic Algorithms, Chapter 4, pages 97–1–25. Springer, 2005.

[94] Eric C. Schnarr, Mark D. Hill, and James R. Larus. Facile: a language and compiler for high-performance processor simulators. *SIGPLAN Not.*, 36(5):321–331, 2001.

[95] Marco sgroi, Luciano Lavagno, Yosinori Watanabe, and Alberto L. Sangiovanni-Vincentelli. Quasi-static scheduling of embedded software using equal conflict nets. In *Proceedings of the 20th International Conference on Application and Theory of Petri Nets*, pages 208–227, London, UK, 1999. Springer-Verlag.

[96] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprette. System design using khan process networks: the compaan/laura approach. volume 1, pages 340–345 Vol.1, Feb. 2004.

[97] Improv Systems. Jazz extensible processor, 2006. `http://www.improvsys.com/`.

[98] Target Compiler Technologies. Chess/checkers, 2006. `http://www.retarget.com`.

[99] M. Thompson, A. D. Pimentel, S. Polstra, and C. Erbas. A mixed-level cosimulation method for system-level design space exploration. In *ESTMED '06: Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, pages 27–32, Washington, DC, USA, 2006. IEEE Computer Society.

[100] Mark Thompson, Hristo Nikolov, Todor Stefanov, Andy D. Pimentel, Cagkan Erbas, Simon Polstra, and Ed F. Deprettere. A framework for rapid system-level exploration, synthesis, and programming of multimedia mp-socs. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 9–14, New York, NY, USA, 2007. ACM.

[101] Anders Sejer Tranberg-Hansen and Jan Madsen. A compositional modelling framework for exploring mpsoc systems. In *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/-software codesign and system synthesis*, pages 1–10, New York, NY, USA, 2009. ACM.

[102] K. Ueda, K. Sakanushi, Y. Takeuchi, and M. Imai. Architecture-level performance estimation method based on system-level profiling. *Computers and Digital Techniques, IEE Proceedings* -, 152(1):12 – 19, jan. 2005.

[103] Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: a survey. *ACM Comput. Surv.*, 29(2):128–170, 1997.

[104] C. Zissulescu, T. Stefanov, B. Kienhaus, and E. Deprettere. Laura: Leiden architecture research and exploration tool. *Field-Programmable Logic and Applications. 13th International Conference, FPL 2003. Proceedings (Lecture Notes in Comput. Sci. Vol.2778)*, pages 911–920, 2003.

[105] Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evol. Comput.*, 8:173–195, June 2000.

[106] Eckart Zitzler and Lothar Thiele. Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach. 1999.

[107] Vladimir D. Zivkovic, Erwin de Kock, Pieter van der Wolf, and Ed Deprettere. Fast and accurate multiprocessor architecture exploration with symbolic programs. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10656, Washington, DC, USA, 2003. IEEE Computer Society.

[108] V. Zivojnovic, S. Pees, and H. Meyr. Lisa-machine description language and generic machine model for hw/sw co-design. In *VLSI Signal Processing, IX, 1996., [Workshop on]*, pages 127 –136, 1996.